Applications of Linear Types

Dhruv C. Makwana
Trinity College

UNIVERSITY OF
CAMBRIDGE

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the
Computer Science Tripos, Part III

University of Cambridge
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: dcm41@cam.ac.uk

May 29, 2018

Declaration

I Dhruv C. Makwana of Trinity College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 10,154

Signed:

Date:

Abstract

In this thesis, I argue that linear types are an appropriate, *type-based formalism* for expressing aliasing, read/write permissions, memory allocation, re-use and deallocation, first, in the context of the APIs of linear algebra libraries and then in the context of matrix expression compilation.

I show that framing the problem using linear types can *reduce bugs* by making precise and explicit, the informal, ad-hoc practices typically employed by experts and matrix expression compilers *and* automate checking them.

As evidence for this argument, I show non-trivial, yet readable, linear algebra programs, that are safe and explicit (with respect to aliasing, read/write permissions, memory allocation, re-use and deallocation) which (1) are more memory-efficient than equivalent programs written using high-level linear algebra libraries and (2) perform just as predictably as equivalent programs written using low-level linear algebra libraries. I also argue *the experience* of writing such programs with linear types is qualitatively better in key respects. In addition to all of this, I show that it is possible to provide such features *as a library* on top of existing programming languages and linear algebra libraries.

Acknowledgements

I would like to thank my family and closest friends for supporting me through my time at Cambridge: you were my strength when I had none. I would also like to thank my supervisors, Dr. Neelakantan Krishnaswami and Dr. Stephen Dolan, for teaching me so much this year. I owe a debt of gratitude to both the 2017 Part III Appeals Committee and those who wrote letters in support of my appeal, for believing I deserved the chance to stay on: never have I valued the privilege of learning what I enjoy and enjoying what I learn more. Lastly, I would like to thank the many staff and research members of the Computer Lab who have been approachable, patient and attentive during my time here.

Contents

1 | Introduction

Linear types allow the compiler and programmer to statically keep track of the resources that a program uses, thus offering a promising solution to the problems associated with complex resource management. However, they have not made their way into many mainstream programming languages, in the same way parametrically polymorphic types have. To illustrate their simplicity and power, I implemented an OCaml library that allows users to learn about and become familiar with linear types, specifically in the context of linear algebra programs.

The main contributions of this thesis are:

- An **original, usable implementation** of a type system that can express aliasing, read/write permissions, memory allocation, re-use and deallocation.
- An **original** demonstration of how that type system can be **applied to the APIs** of linear algebra libraries and the **benefits** of doing so.
- Many **new** examples of how that type system can **automatically check** for common aliasing, read/write permission, memory allocation, re-use and deallocation **errors** in the context of linear algebra programs.
- An **original** demonstration of how that type system can be **used** for **matrix expression compilation**.
- **New and readable implementations** of **non-trivial** linear algebra programs that **take advantage** of said type system.
- A **new solution** to the **dichotomy** of **readability**, **ease of reasoning and safety** of high-level linear algebra libraries versus the **memory-efficiency** of low-level linear algebra libraries.
- A **new library design** to provide said solution in a way that **integrates well** with existing OCaml code and linear algebra libraries.

2 | Background

I will outline the concept of linear types and show how they can be used to solve the problems faced by programmers writing code that uses linear algebra libraries. I will emphasise *practical* and intuitive explanations of linear types to keep this thesis accessible to programmers as well as academics not familiar with type theory and will give only a terse overview of the history and theory behind linear types for the interested reader to pursue further.

$$\frac{}{\Gamma, x : t \vdash x : t; \Gamma}\ \textsc{Var} \qquad \frac{\Gamma_1 \vdash e_1 : t_1; \Gamma_2 \qquad \Gamma_2 \vdash e_2 : t_2; \Gamma_3}{\Gamma_1 \vdash (e_1, e_2) : t_1 \otimes t_2; \Gamma_3}\ \textsc{Pair}$$

Figure 2.1 – A typing rule has the general form $\Gamma_{\text{in}} \vdash e; \Gamma_{\text{out}}$. Typing a variable *removes* it from the environment. Typing a pair requires that each component of the pair uses *different* resources from the environment.

2.1 Tracking Resources with Linearity

Familiar examples of using a type system to express program invariants are existential types for abstraction and encapsulation, and polymorphic types for parametricity and composition (a.k.a generics). Less-known examples include dependent types for contracts or pre- and post-conditions. The advantages of using a type system to express program invariants are summarised by saying: the stronger the rules you follow, the better the guarantees you get about your program *before* you run it. At first, the rules may seem restrictive, but similar to how the rules of grammar, spelling and more generally writing help a writer communicate their ideas more clearly, so too do typing rules make it easier for the programmer to communicate the intent and assumptions of their programs. An added, but often overlooked benefit is automated-checking: a programmer can boldly refactor their programs and the type checker will *assist* in enforcing the invariants it was given by highlighting where they were violated.

Linear types are a way to help a programmer track and manage resources. In practical programming terms, they enforce the restriction that a value may be used exactly once[1] (Figure 2.1 shows two example typing rules for a typical, linearly typed language). While this restriction may seem limiting at first, precisely this constraint can be used to express common invariants of typical programs. For example: a file or a socket, once opened *must* closed; all memory that is manually allocated *must* be freed. C++'s destructors and Rust's Drop-trait (and more generally, its borrow-checker) attempt to enforce these constraints by basically doing the same thing: any resource that has not been moved is deallocated at the end of the current lexical scope. Notably, these languages also permit aliasing, alongside rules enforcing when it is acceptable to do so. On face value, the above one-line description of linear types prevents aliasing or functions such as $\lambda x.\ (x, x)$; such features are still allowed (albeit in a more restricted fashion) in a *usable* linear type system designed for working with linear algebra libraries.

[1]This definition may differ from more colloquial uses in discussions surrounding *substructural* type systems and/or Rust.

```
# Numpy (Python)
import numpy.matlib
a = [[1,0],[0,1]]
b = [[4,1],[2,2]]
c = numpy.matmul(a,b)
# Julia
c = [1 0; 0 1] * [4 1; 2 2]
```

```
(* Owl (OCaml) *)
open Owl
let a = Mat.of_arrays
  [|[|1.;0.|];[|0.;1.|]|]
let b = Mat.of_arrays
  [|[|4.;1.|];[|1.;2.|]|]
let c = Mat.(a *@ b)
```

Figure 2.2 – Matrix Multiplication in Numpy (Python), Julia and Owl (OCaml).

2.2 Problem in Detail

Given this background, the most pertinent question at hand is: what problems do linear algebra library users (and implementors) typically face? The answer to this question depends on which of two buckets a programmer falls (or is forced by domain) into. On one side, we have users of high-level frameworks such as Owl (for OCaml), Julia and Numpy (for Python); on the other, we have users of more manual, low-level libraries such as BLAS (Basic Linear Algebra Subroutines) for languages like C, C++ and Fortran.[2] Most of what follows applies to *dense* linear algebra computations rather than *sparse* ones because for the latter, memory allocated for results depends on the sparsity of the inputs and so is not immediately amenable to the techniques proposed in this thesis.

2.2.1 One Too Many Copies and a Thousand Bytes Behind

In Figure 2.2, we see that matrix multiplication is fairly trivial to write and execute in Numpy, Julia and Owl. Let us call this approach *value-semantic*, meaning that objects are *values* just like integers and floating-point numbers. This approach confers two key advantages to the programmmer: it is easy to read (equational and algebraic expressions) and it is easy to reason about (as one would with a mathematical formula). Although this approach does permit *aliasing*, the consequences are benign because the result of any computation is a *new* value, distinct from any used during the calculation of that value.

However, these advantages come with some costs: constantly producing new values is wasteful on memory (although the example given in Figure 2.2 is only a 2×2

[2]I am not including Rust in this comparison because its linear algebra libraries are under active development and not as well-known/used. Later on, I will compare and contrast Rust (a language with in-built support of substructural features to track resources) with this project to evaluate the classic (E)DSL-versus-language-feature debate as it applies to linear types.

```
let mul x y =
  if same_shape x y then
    let y = copy y in
    (_owl_mul (kind x) (numel x) x y y; y)
  else
    broadcast_op (_owl_broadcast_mul (kind x)) x y
```

Figure 2.3 – Implementation of Matrix Multiplication in Owl (OCaml). Note the 'copy' for the result and the unsafe '_owl_mul' operation used to perform an in-place multiplication.

matrix, many real-world datasets can contain up to gigabytes of data). A complex expression may create many short-lived temporaries which would need to be reclaimed by a garbage-collector (see Figure 2.3). Libraries taking a *value-semantic* approach offer a dichotomy for a user wishing to implement a new algorithm: either use the existing and safe primitives to build an easy to reason about but slower, more memory-intensive algorithm, or use escape-hatches (typically provided by most libraries, which permit in-place modification of objects) to build faster, less readable and more memory-efficient algorithms which are harder to reason about.

2.2.2 IHNIWTLM

The title of this subsection[3] illustrates the problem with low-level libraries: readability (equational and algebraic expressions) and ease of reasoning is sacrificed at the altar of performance and (memory) efficiency.

Although escape-hatches do exist in high-level libraries, their use is discouraged. Systematic consideration of performance requires lowering the level of abstraction a programmer is working on. At this level, **several** factors such as memory layout, allocation, re-use as well as cache behaviour and parallelism are important. Of these, **only memory allocation and re-use** are relevant to linear types and this thesis.

In Fortran (Figure 2.4) and C, temporary storage for all intermediate values must be managed by the programmer. While this approach leads to verbose, less readable and harder to reason about code, the explicitness is good for understanding the memory concerns of the program.

[3] I Have No Idea What Those Letters Mean.

```
      program blasMatMul
      implicit none
      real*4 a(2,2), b(2,2), c(2,2)
C External from BLAS
      external dgemm
C Initialize in column major storage of Fortran
      data a/ 1,0,0,1/
      data b/ 4,1,1,2/
C               tfm  tfm  rowA colB K  alpha a lda  b  ldb beta c  ldc
      call dgemm('N', 'N', 2,   2,   2, 1.0,  a,  2, b, 2,  0.0, c, 2)
```

Figure 2.4 – One of *several* BLAS (Fortran) routines for Matrix Multiplication.

On the other hand, C++ (with operator overloading) can end up looking fairly readable. For safety and correctness, expressions are typically handled with value-semantics (the result of any computation is a *new* value, distinct from any used during the calculation of that value). However, given *extra* information about aliasing (Eigen, Figure 2.5) or usage of intermediate expressions (uBLAS, Figure 2.6), the number of temporaries allocated can be reduced and increased *implicitly* to improve performance (remove unnecessary allocations or re-calculations respectively). Further tricks to improve performance include expression templates (building up an expression-tree at compile time and then pattern-matching on it to produce code) and lazy evaluation (only calculating a result when it is needed). These will be discussed in more detail in Chapter 5.

It is important to note that should a `noalias` annotation be wrong, the program's behaviour is very likely to end up being undefined (like how `memcpy()` for overlapping regions of memory is explicitly undefined in the POSIX and C standards). Indeed, one of Fortran's strengths lies in assuming that references cannot be aliased (with certain caveats) in more cases than C permits (this informal, general statement comes with many nuances left for the interested reader to pursue).

2.3 Proposed Solution

My proposed solution to this dichotomy (readability, ease of reasoning and safety versus memory-efficiency) is a *domain-specific language* (DSL), called LT4LA (Linear Types for Linear Algebra). It is written in OCaml and transpiles to OCaml. It offers readable, explicit management of aliasing, read/write permissions, memory allocation, re-use and deallocation all with automated checking: offering a

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
int main()
{
    Eigen::Matrix2d a,b,c;
    a << 1, 0, 0, 1; b << 4, 1, 1, 2; c << 0, 0, 0, 0;
    a * b; // new matrix
    c += a * b; // temporary for correctness in case of aliasing
    c.noalias() += a * b; // no temporaries
}
```

Figure 2.5 – Some examples of Matrix Multiplication in Eigen. Using expression templates (to be discussed later) and *explicitly provided* aliasing information, Eigen can emit a single BLAS 'dgemm'-like call for the last line, mirroring the Fortran example of Figure 2.4.

```
noalias(C) = prod(A, B);
// Preferable if T is preallocated
temp_type T = prod(B,C); R = prod(A,T);
prod(A, temp_type(prod(B,C));
prod(A, prod<temp_type>(B,C));
```

Figure 2.6 – Boost uBLAS example of Matrix Multiplication. Temporaries need to be marked as such to prevent unnecessary re-computation of values.

safety net to catch the baby whilst swiftly disposing of the bath water. Although for expository and testing purposes I have defined a concrete-syntax, a full implementation would make use of a language's syntax-extension features (such as PPX for OCaml) to *embed* the DSL into the host language. Such an embedding is straightforward but fairly tedious to implement. As a half-way point, I used compile-time code generation to make the DSL's output available to OCaml for testing and evaluation.

Let us have a look at a few examples of functions we can write with linear types. We can define the factorial function (Figure 2.7) and sum over an array (Figure 2.8).

The syntax is intended to resemble OCaml's, apart from the spurious `'!'`s found here and there (they are annotations to show that we can use a value more than once). In Figure 2.8, we see `row` has type (`'x arr`). More detailed explanations of *what and why* will be given in Chapter 3, but for now, it is enough to know it means we can only *read* from `row`, and not write to it. If we did try to write to or free `row`, we would get a helpful error message, as shown in Figure 2.9.

Now suppose we are trying to square a matrix, using a 'dgemm' like BLAS routine

```
let rec f ( !x : !int ) : !int =
    if x < 0 || x = 0 then
        1
    else
        x * f (x - 1) in f
;;
```

Figure 2.7 – Factorial function in LT4LA.

```
let rec f (!i : !int) (!n : !int) (!x0 : !elt)
          ('x) (row : 'x arr) : 'x arr * !elt =
    if i = n then
        (row, x0)
    else
        let (row, !x1) = row[i] in
        f (i + 1) n (x0 +. x1) 'x row in
    f
;;
```

Figure 2.8 – Summing over an array in LT4LA.

```
let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality:                                          *)
(*     z arr                                                         *)
(* with                                                              *)
(*     'x arr                                                        *)
(*                                                                   *)
(* Var 'x is universally quantified                                  *)
(* Are you trying to write to/free/unshare an array you don't own?   *)
(* In test/examples/sum_array.lt, at line: 7 and column: 19          *)
```

Figure 2.9 – Attempting to write to or free a read only array in LT4LA.

```
let (a1, a2) = share _ a in
let ((a1, a2), c) = simple_dgemm _ a1 _ a2 c in
let a = unshare _ a1 a2 in
let () = free a in c
```

Figure 2.10 – Squaring a matrix in LT4LA (assuming `c` is initially 0).

(called 'simple_dgemm'[4] in Figures 2.10 and 2.11) which takes two read-only matrices and a third matrix it can write to and performs $C := AB + C$. How would we use such a routine to square a matrix, assuming $C = 0$ initially? Surely this would break linearity, since A would have to be passed in to the function twice, and we can only use non-!-annotated variables once?

To solve this, we use a special primitive called `share` to produce (more) read-only *aliases* of any matrix. We then pass these into the function, and it works as expected. Once the squared matrix has been obtained, we may not want the original anymore, and thus decide to free it. Before we do so, we must first `unshare` any read-only aliases that exist, to get back a single, read-write handle.

If we tried to free one of the read-only aliases before or instead of `unshare`-ing them, then we would get the error shown in Figure 2.11. Briefly, a `z arr` is a read-write array, everything else (`'x arr` or `_ s arr`) is read-only. The types of `free`, `share` and `unshare` (see Appendix B) are set up so that you can only free something when you have read-write access to it, guaranteed by linearity to be the only name in scope with this capability (aliases can only be read-only). Conversely, if we *forgot* to `free a`, we would also get `Variable a not used` error.

Another way in which 'simple_dgemm' may be misused is by instantiating its informal description of $C := AB + C$ with $B = C = A$ and so mistakenly concluding that it computes $A := A^2 + A$ in-place. However, the type of `share` prevents this as well – `let (a11, a12) = share _ a1 in simple_dgemm _ a11 _ a12 a2` would result in an error similar to the one shown in Figure 2.11.

2.4 Further Reading and Theory

No exposition of linear types would be complete without a mention of Girard's Linear Logic [1]. As mentioned in the Stanford Encyclopedia of Philosophy, it is "a refinement of classical and intuitionistic logic. Instead of emphasizing truth, as in

[4]For the purposes of this example, I assume 'simple_dgemm' has type `'x . 'x mat --o 'y . 'y mat --o z mat --o ('x mat * 'y mat) * z mat`.

```
let (a1, a2) = share _ a in
let ((a1, a2), c) = simple_dgemm _ a1 _ a2 c in
let () = free a1 in c
(* Error:                                                          *)
(* Could not show equality:                                        *)
(*     z arr                                                       *)
(* with                                                            *)
(*     z s arr                                                     *)
(*                                                                 *)
(* Could not show z and z s are equal.                             *)
(* Are you trying to write to/free an array before unsharing it? *)
(* In test.lt, at line: 3 and column: 17                           *)
```

Figure 2.11 – Attempting to free a read-only alias of matrix.

classical logic, or proof, as in intuitionistic logic, linear logic emphasizes the role of formulas as resources." A walk from logic to programming along the well-trodden Curry-Howard bridge brings us to linear types [2].

For the category theory inclined reader, the !-operator (sometimes, for reasons elided here, called *exponentiation*) forms a co-monad; for the rest of us, this entails two (rather simple) facts about a value you can use any number of times: you can (1) can use it once (co-unit), and (2) can pass it to many contexts that will use it many times (co-multiply).

More generally, by annotating variables in the context with their usage (when implementing a type checker for a linearly typed language), we can express the rules of *substructural* (including affine, relevant and ordered type systems) under the more general framework of *co-effects* [3].

Stepping further back, both the practice and theory behind resource-aware programming has made visible progress in the past few years. On the programming side, we have Linear Haskell, Rust and Idris (experimental). On the research side, we have Resource Aware ML [4] and the tantalising promise of integrating linear and dependent types [5].

2.5 Summary

I have given an *intuitive* exposition of linear types with *fractional-capabilities* [6], emphasising small, but illustrative and practical code examples, leaving *what* is going on and *why* it works as details for the next chapter. I have shown that it is possible to solve the dichotomy of readability, ease of reasoning and safety versus memory-efficiency by providing explicit and automatically checked management of aliasing, read/write permissions, memory allocation, re-use and deallocation with a *type system*. This prevents a **whole class** of errors that can occur with low-level languages *at compile time*. For example, **the types make it impossible to use 'simple_dgemm' incorrectly**. I demonstrated these features using the context of linear algebra libraries as as a specific example. In the next chapter, I will show that these features can be further applied to the problem of *matrix expression compilation*. I will also explain how to express and implement them so that they can be provided and used *as a library*.

3 | Implementation

I will describe the structure of LT4LA: a DSL (domain specific language) **library** written in OCaml that transpiles to OCaml. I will explain the features of its core language and show how readable (equational and algebraic expressions) and easy to reason about linear algebra programs can be elaborated into memory-efficient ones in the core language that are then checked for safety (with respect to aliasing, read/write permissions, memory allocation, reuse and de-allocation). Finally, I will explain how such programs can be transpiled to OCaml code that is not obviously safe. Although I refer to OCaml-specific features of the type checker and transpiler, I believe the ideas described in this chapter can easily be implemented in other languages and are also general enough to transpile to other back-end languages, such as C or Fortran.

3.1 Structure of LT4LA

LT4LA follows the structure of a typical compiler for a (E)DSL. From the start, I made a concerted effort to (1) write pure-functional code (typically using a monadic-style) which helped immensely with modularity and debugging when tests showed errors (2) produce readable, useful and precise error-messages in the hope that someone who did not understand linear types could still use LT4LA (3) write tests and set-up continuous-integration for all non-trivial functions so that I could spot and correct errors that were not caught by OCaml's type system whenever I implemented new features or refactored my code.

1. **Parsing**. A generated, LR(1) parser parses a text file into a syntax tree. I tried to mimic OCaml syntax modulo a few extensions and keywords so that it is familiar and thus easy to pick-up for OCaml users. In general, this part will vary for different languages and can also be dealt with using combinators or syntax-extensions (the EDSL approach) if the host language offers such support .

2. **Desugaring**. The syntax tree is then desugared into a smaller, more concise, abstract syntax tree. This allows for the type checker to be simpler to specify and easier to implement.

3. **Matrix Expressions** are also desugared into the abstract syntax tree through simple, yet effective pattern-matching.

4. **Type checking**. The abstract syntax tree is explicitly typed, with some inference to make writing typical programs more convenient.

5. **Code Generation**. The abstract syntax tree is translated into OCaml, with a few 'optimisations' to produce more readable code. This process is type-preserving: LT4LA's type system is embedded into OCaml's (Figure 3.1), and so the OCaml type checker acts as a sanity check on the generated code.

6. **Executable Artifacts**. A transpiler and a REPL are the main artifacts produced for this thesis. For evaluation, I implemented Kalman filters in Owl, LT4LA and CBLAS/LAPACKE and a benchmarking program to measure execution times.

7. **Tests**. As mentioned before, almost all non-trivial functions have tests to check their behaviour. The output of the transpiler was also tested by having

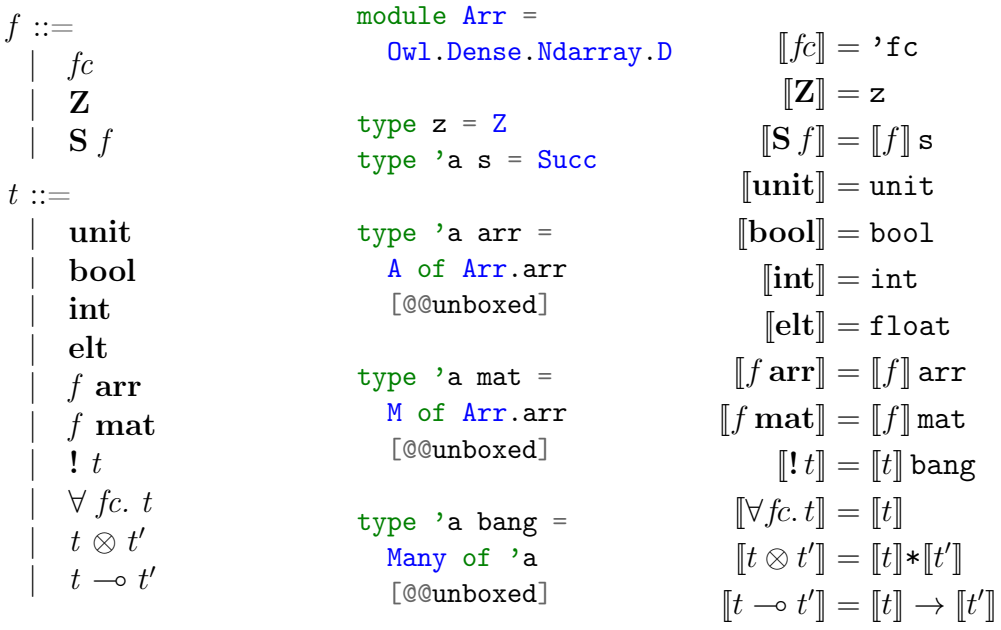

Figure 3.1 – Type grammar of LT4LA (left) and my translation of it into OCaml (right).

the build system generate OCaml code at compile time, which in turn could then be compiled and tested like handwritten OCaml code.

3.2 Core Language

A full description of the core language can be found in Appendix A. For convenience, I have reproduced its type grammar and its translation into OCaml in Figure 3.1. The point of LT4LA is to keep the same values and types we are familiar with from OCaml, but *restrict their usage*. LT4LA's core language's main features are: intuitionistic values, value-restriction, fractional-capabilities (inferrable at call sites), if-expressions and recursion.

3.2.1 Intuitionistic Values

To make LT4LA a usable DSL, I needed a way of allowing values to either never be used or be used more than once. For this, I added the !-constructor at the type-level of the DSL and its corresponding introduction (the `Many`-constructor) and elimination (the `let Many <id> = .. in ..` destructor) forms at the term-level of the DSL. The idea behind the !-constructor is that a value of that type uses no 'resources' (linearly-typed expressions).

To start off with, it is enough to say that anything which can be passed around by copying, will have a !-type. This includes integers, elements and booleans. So `3 : !int` and `3. +. 4. : !elt`. However, all bindings are still linear by default, so to emulate intuitionism, I desugar `let !x = <exp> in <body>` to `let Many x = <exp> in let Many x = Many (Many x) in <body>` (similarly for function argument bindings). The reader can check (using the rules in Appendix A) that this has the effect of moving `x : !t` from the linear to the intuitionistic environments, only if `<exp> : !t`.

However, just that desugaring alone is not enough to prevent a user from taking an array or matrix and moving it into the intuitionistic environments. Why? There are certain situations in which we *should not* use the `Many` constructor. Consider the following code: `let Many x = Many (array 5) in <body>`; the expression `array 5` uses no linearly-typed variables from the environment. Although we could just reject types of the form `!(_ arr)` to fix this simple example, what about pairs `let Many xy = Many (3, array 5) in <body>`? Ad-hoc pattern matching on the type cannot account for all possible situations. With the last case, we can use `xy` as many times as we would like, destruct the pair to get the second component and thus create *distinct* read-write aliases to the same array. Alas, now arrays can be used intuitionistically and all the benefits of linearity are lost. Or are they?

3.2.2 Value-Restriction

Not quite, but to understand how we can fix this problem, we need to question an assumption left implicit up until this point: what does `Many` even mean in the DSL? How is it translated to OCaml? What is the OCaml runtime behaviour of the DSL's `Many`-constructor? One option is to go down the C++ route and make `Many` behave like a `shared_ptr` and act as a runtime reference-count for arrays. I chose to not go for this option because it went against the *explicitness* and *predictability* that C and Fortran have. It would make analysing when and what is allocated and freed more like the higher-level languages I was trying to move away from.

My aim is to show linear types are simple to understand and apply, enough so that they can be grafted (in a limited way) on to existing languages as a *library*. In that spirit, the simplest thing that the DSL's `Many`-constructor can mean in the OCaml runtime is *nothing*. LT4LA's !/`Many` constructors are translated into OCaml `bang`/`Many` constructors declared as `type 'a bang = Many of 'a [@@unboxed]`. The

unboxed annotation means that the type and its constructor only exist for the purpose of type checking in OCaml; *the OCaml runtime representation of values of type* `'a` *and* `'a bang` *is exactly the same.*

With this understanding, our problem is that arrays and matrices are unlike other values such as integers and elements because (under the OCaml hood) calling a function with an array argument copies a *pointer* to the array rather than the array itself. So, we can start making a distinction, *defining* elements, integers, booleans, intuitionistic variables, units and lambda-expressions that capture no linear variables as *values* (since they cannot break referential-transparency) and anything else (arrays, matrices, expressions which can be reduced, such as function application and if-expressions) as not being 'values'. If this sounds familiar, it is because this is the same *value-restriction* 'trick' from the world of polymorphic types applied to linearity instead. We then have the rule that in LT4LA, we can only use `Many` on expressions that are defined to be *values* and *use no linear variables.*

3.2.3 Fractional-Capabilities and Inference

I started off with linearity and showed how it helps a programmer keep track of memory allocation and deallocation, and then demonstrated how to add intuitionism for the only values we wish to be intuitionistic. I will now explain how I enforce safe aliasing and read/write permissions.

Array and matrix types are parameterised by *fractional-capabilities* [6]. A fraction of 1 (2^0) represents complete ownership of a value. Creating an array gives you ownership of it; the function `array : !int --o z arr` (where `z` represents '0'). Once you have ownership of an array, you can free it: `free : z arr --o unit`. Importantly, because a linear value may only be used once, the array just freed is *out of scope* for following expressions, **preventing use-after-free**. Ownership also enables you to write to the array: `set : z arr --o !int --o !elt --o z arr` (the syntax `w[i] := j` is just sugar for `set w i j`). Here, linearity prevents accessing aliases which represented the array *before* the mutation.

Any fraction less than 1 (for simplicity, limited to 2^{-k} in this system, for a positive integer k) represents read-only access. So, the `'x` represents a natural number (either a zero `z`, variable `'x` or a successor ($+1$) of a natural number). Hence, you can read from (index) any array `get : 'x . 'x arr --o !int --o 'x arr * !elt` (the syntax `let !v <- w[i]` is just sugar for `let (w, !v) = get _ w i`). In gen-

eral, a left-arrow `<-` signifies transparent rebinding with returned values: it means a program can *appear* to use a variable multiple times, important for keeping LT4LA usable and readable. The underscore is how a programmer tells the compiler to automatically *infer* the correct fractional-capability based on the other arguments passed to the function. In conjunction with the requirement that function declarations need type-annotations for their arguments, **this allows fractional-capabilities to be *correctly inferred* in *any* LT4LA expression**.

Fractions exist to provide both safe aliasing and read/write permissions, via the primitives `share : 'x . 'x arr --o ('x s) arr * ('x s) arr` and `unshare: 'x . ('x s) arr --o ('x s) arr --o 'x arr`. For the former, the two arrays returned (which are just aliases of the given array) can now only be read from and not written to. If you want to write to this array, you must use the latter to combine other read-only aliases until you are left with only one value of type `z arr`, guaranteeing no other aliases exists.

Given this set-up, a programmer now *statically* has *perfect* information about aliasing and ownership of values in the program. They can only write to or free an array only when they own it; ownership guarantees no other aliases exist. In Figure 3.5, I show how this perfect information can be used to write more readable (equational and algebraic) code using easy to reason about, value-semantic expressions whose *intensional* behaviour and assumptions (regarding aliasing, read/write permissions and memory allocation, re-use and de-allocation) are also explicitly and precisely described and *automatically checkable*. Now the programmer need not resort to manually figuring out and inserting `noalias` annotations and worrying about what variables can and cannot be written to, re-used or freed; instead they can let the loyal and tireless compiler do the heavy lifting.

3.2.4 If-Expressions

An if-expression's condition may evaluate either way during execution, so a programmer must guarantee that both branches use the same set of linear variables. Writing the type checker in a pure-functional monadic style paid off here because I could now sandwich monadic values with state adjustments either side of it. Given two monadic values that represented type checking two branches of an if-expression, I could use the code in Figure 3.2 to easily save, reset and compare the state either side of running those monadic values.

```
let same_resources (wf_a, loc_a) (wf_b, loc_b) =
  let open Let_syntax in
  (* Save state *)
  let%bind {used_vars=prev; env=old_env; _} as state = get in
  (* Reset, run a, save state *)
  let%bind () = put { state with used_vars = empty_used } in
  let%bind res_a = wf_a in
  let%bind {used_vars=used_a; _} as state = get in
  (* Reset, run b, save state *)
  let%bind () = put { state with used_vars = empty_used; env = old_env } in
  let%bind res_b = wf_b in
  let%bind {used_vars=used_b; _} as state = get in
  (* Check if same resources *)
  let keys_a, keys_b = (* convert to (used_a, used_b) to sets *) in
  if Set.equal keys_a keys_b then
    (* merge used_vars and used_b environments *)
  else
    (* report differences *)
```

Figure 3.2 – Implementation of same_resources helper method for type checking if-expressions. Note the monadic style helped compose computations that affected the type checker's state in a simple manner.

3.2.5 Functions and Recursion

A non-recursive function may be used more than once if it does not refer to any linear variables from the surrounding scope. So, we can desugar something like `let x = 3 in let !f (y : !int) = x + y in <body>` to `let x = 3 in let Many f = Many (fun y : !int -> x + y) in <body>`. Recursion is slightly more complicated: we can desugar the factorial function (Figure 2.7) to `let Many f = fix (f, x : !int, if (*..*) : !int) in <body>`. However, `fix`, like `Many`, must also not use any linear variables from its surrounding scope, because a recursive function may evaluate its body multiple times.

```
subroutine kalman(mu, Sigma, H, INFO, R, Sigma_2, data, mu_2, k, n)
implicit none

integer, intent(in)    :: k, n
real*8,  intent(in)    :: Sigma(n,n), H(k,n), mu(n)
real*8,  intent(inout) :: data(k)       ! data, H*mu - data , (H*Sigma*H^T + R)^-1*(H*mu - data)
real*8,  intent(inout) :: R(k, k)       ! R, H*Sigma*H^T + R
integer, intent(out)   :: INFO          ! INFO
real*8,  intent(out)   :: Sigma_2(n,n) ! H^T*(H*Sigma*H^T + R)^-1*H, Sigma, Sigma*(I - H^T*(H*Sigma*H^T + R)^-1*H*Sigma)
real*8,  intent(out)   :: mu_2(n)       ! mu, Sigma*H^T*(H*Sigma*H^T + R)^-1*(H*mu - data) + mu
real*8                 :: H_2(k,n)      ! H * Sigma , H , (H*Sigma*H^T + R)^-1*H
real*8                 :: chol_R(k,k)   ! R, U where (H*Sigma*H^T + R)=U^T*U
real*8                 :: H_data(n)     ! H^T*(H*Sigma*H^T + R)^-1*(H*mu - data)
real*8                 :: N_N_tmp(n,n) ! H^T*(H*Sigma*H^T + R)^-1*H*Sigma

call dsymm('R', 'U', k, n, 1, Sigma, n, H, n, 0, H_2, n)            ! H_2     := 1. * H   * Sigma +  0. * H_2
call dgemm('N', 'T', k, k, n, 1, H_2, n, H, n, 1, R, k)             ! R       := 1. * H_2 * H     +  1. * R
call dgemm('N', 'N', k, 1, n, 1, H, n, mu, 1, -1, data, 1)          ! data    := 1. * H   * mu    + -1. * data
call dcopy(k*n, H, 1, H_2, 1)                                       ! H_2     := H
call dcopy(k*k, R, 1, chol_R, 1)                                    ! chol_R  := R
call dposv('U', k, n, chol_R, k, H_2, n, INFO)                      ! chol_R  := U where R = U^T * U
                                                                    ! H_2     := R^-1 * H_2
call dpotrs('U', k, 1, chol_R, k, data, 1, INFO)                    ! data    := R^-1 * data
call dgemm('T', 'N', n, n, k, 1, H, n, H_2, n, 0, Sigma_2, n)       ! N_N_tmp := 1. * H^T * H_2    + 0. * N_N_tmp
call dgemm('T', 'N', n, 1, k, 1, H, n, data, 1, 0, H_data, 1)       ! H_data  := 1. * H^T * data   + 0. * H_data
call dcopy(n, mu, 1, mu_2, 1)                                       ! mu_2    := mu
call dsymm('L', 'U', n, 1, 1, Sigma, n, H_data, 1, 1, mu_2, 1)      ! mu_2    := 1. * Sigma * H_data  + 1. * mu_2
call dsymm('R', 'U', n, n, 1, Sigma, n, Sigma_2, n, 0, N_N_tmp, n)  ! N_N_tmp := 1. * N_N_tmp * Sigma   + 0. * N_N_tmp
call dcopy(n**2, Sigma, 1, Sigma_2, 1)                              ! Sigma_2 := Sigma
call dsymm('L', 'U', n, n, -1, Sigma, n, N_N_tmp, n, 1, Sigma_2, n) ! Sigma_2 := -1 * Sigma * N_N_tmp + 1. * Sigma_2

RETURN
END
```

Figure 3.3 – Kalman filter in Fortran 90.

$$\mu' = \mu + \Sigma H^T (R + H\Sigma H^T)^{-1}(H\mu - \text{data})$$
$$\Sigma' = \Sigma(I - H^T(R + H\Sigma H^T)^{-1}H\Sigma)$$

Figure 3.4 – Kalman filter equations (credit: matthewrocklin.com).

3.3 Matrix Expressions

We have now arrived at an extended application of linear types. I will show how we can apply the ability to automatically check aliasing, read/write permissions, memory allocation, re-use and deallocation, to the domain of matrix-expression compilation (automatically generating code like Figure 3.3 from mathematical expressions like Figure 3.4).

In Figure 3.3, we see the difficulty of efficiently implementing a *Kalman filter* [7], a powerful set of equations (Figure 3.4) applicable to a wide variety of problems. From the comments, we see that every variable is annotated with the matrix expression that it will hold at some point during the computation (an equivalent alternative, say in C++, could be to have a meaningful name for each step and manually keep track of which names alias the same location).

In contrast, Figure 3.5, offers the advantages of

- aliasing: labelling each step with a different, more meaningful variable name,
- easily spotting which resources are being passed in and which are allocated for the function (new/copy),
- unambiguously seeing *when* and what values are freed or written-over;

and have the compiler automatically ensure the safety of each of the above by respectively

- making it impossible to refer to values which are no longer usable,
- ensuring all values are declared and *initialised* correctly before they are used,
- checking values are neither used after they are freed/written-over *nor* leaked.

Indeed, an inexperienced programmer could take the naïve approach of just copying sub-expressions by default and then letting the compiler tell them which copies are never used and removing them systematically until their program type checks. While it is not quite a black-box, push-button compilation of an expression, I

```
      let !kalman
        ('s) (sigma : 's mat) (* n,n *)
        ('h) (h : 'h mat)     (* k,n *)
        ('m) (mu : 'm mat)    (* n,1 *)
        (r_1 : z mat)         (* k,k *)
        (data_1 : z mat)      (* k,1 *) =
        let (h, (!k, !n)) = sizeM _ h in
(*16*)  let sigma_h <- new (k, n) [| h * sym (sigma) |] in
(*17*)  let r_2 <- [| sigma_h * h^T + r_1 |] in
(*18*)  let data_2 <- [| h * mu - data_1 |] in
(*19*)  let (h, new_h) = copyM_to _ h sigma_h in
(*20*)  let new_r <- new [| r_2 |] in
(*21*)  let (chol_r, sol_h) = posv new_r new_h in
(*23*)  let (chol_r, sol_data) = potrs _ chol_r data_2 in
        let () = freeM (* k,k *) chol_r in
(*24*)  let h_sol_h <- new (n, n) [| h^T * sol_h |] in
        let () = freeM (* k,n *) sol_h in
(*25*)  let h_sol_data <- new (n, 1) [| h^T * sol_data |] in
(*26*)  let mu_copy <- new [| mu |] in
(*27*)  let new_mu <- [| sym (sigma) * h_sol_data + mu_copy |] in
        let () = freeM (* n,1 *) h_sol_data in
(*28*)  let h_sol_h_sigma <- new (n,n) [| h_sol_h * sym(sigma) |] in
(*29*)  let (sigma, sigma_copy) = copyM_to _ sigma h_sol_h in
(*30*)  let new_sigma <- [| sigma_copy - sym (sigma) * h_sol_h_sigma |] in
        let () = freeM (* n,n *) h_sol_h_sigma in
        ((sigma, (h, (mu, (r_2, sol_data)))), (new_mu, new_sigma)) in
      kalman
      ;;
```

Figure 3.5 – Kalman filter in LT4LA. In contrast with the Fortran (Figure 3.3) or C (Figure 4.1) implementations, this one is readable (equational and algebraic), using easy to reason about, value-semantic expressions. In addition to that, its *intensional* behaviour and assumptions (regarding aliasing, read/write permissions and memory allocation, re-use and de-allocation) are also explicitly and precisely described and *automatically checkable*.

would argue it is just as easy (if not easier) to learn how to work with as Rust's borrow checker.

3.3.1 Elaboration

All of the syntax in Figure 3.5 can be unambiguously desugared *before* type checking, through fairly simple pattern matching. Matrix expression compilation is well-trodden territory in academia [8, 9, 10, 11, 12] but this is, to my knowledge, the **first type-based approach** to it.

An overview of the translations are in Figure 3.6; details about choosing between

```
let x <- [| y |] in ..              ==> let (y,x) = copyM_to _ y x in ..
let x <- new [| y |] in ..          ==> let (y,x) = copyM _ y in ..
let x <- [| a*b + c |] in ..        ==> let x <- [| 1.*a*b + 1.*c |] in ..
let x <- [| i*a*b + j*c |] in ..    ==> let ((a,b), c) = (* BLAS *) in ..
let x <- new (m, n) [| a*b |] in .. ==> let c = matrix m n in
                                        let x <- [| 1.*a*b + 0.*c |] in ..
```

Figure 3.6 – Syntactic translations of matrix expressions to linearly-typed matrix functions. Further annotations on the matrix variables (`sym` or `^T`) determine which BLAS routine is called and what parameters it is passed.

'symm' or 'gemm' are omitted for brevity. Before settling on this approach, I tried implementing a more general type-directed, nested matrix expression compiler; I will now highlight some of the difficulties inherent in the problem.

One of the first hurdles I encountered was compositionality: to compile $AB + C$ it is typically better to use a BLAS routine directly rather than first compiling A, then B, then adding a call to multiply them, then compiling C and finishing with a call to add the results.

Another compositionality problem is that a call to a linear function does not just return a result, but a sequence of re-bindings which dictate which variables are still usable (in scope). As such, to compile an expression, you need to provide a CPS-style function of type $var \to exp$ representing how you would use a variable representing the result in the rest of the computation.

However, the type of that variable also determines how you can use it: can you write to it or must you copy it? This information depends on how the expression representing the variable was elaborated and adds to the complexity of the pattern-matching.

Copying leads us into dealing with temporaries: do you first allocate all temporaries in new matrices (SSA-style) and then analyse dimensions to figure out which slots of memory can be re-used via copy coalescing? Or do you try and infer the live ranges of available resources from the environment as you go? Can, and should, you type arrays and matrices (or n-dimensional tensors) as the same?

Adding in more and more considerations, the problem starts to resemble register allocation: there are registers of different types and sizes, many (non-orthogonal) instructions to choose from, a cost model to take into account all whilst trying to balance the number of registers in use and the number of instructions emitted.

```
let rec f i n x0 row =
  if Prim.extract @@ Prim.eqI i n then (row, x0)
  else
    let row, x1 = Prim.get row i in
    f (Prim.addI i (Many 1)) n (Prim.addE x0 x1) row
in
f
```

Figure 3.7 – Recursive OCaml function for a summing over an array, generated (at *compile time*) from the code in Figure 2.8, passed through `ocamlformat` for presentation.

3.4 Code Generation

Code generation is a straightforward mapping from core LT4LA constructs to OCaml constructs, with the addition of `Many` constructors to wrap integer, element and boolean literals.

To make the code produced readable, I added a few 'optimisations', to the compiler, to 're-sugar' some of the constructs translated where appropriate. Almost all of them involved the erasing the !-eliminator which is not needed in regular, intuitionistic OCaml. So, we can simply replace any expression of the form:

- `let xy, z = <exp> in let x,y = xy in <body>` with `let (x,y), z = <exp> in <body>`,
- `let Many x = x in let Many x = Many (Many x) in <body>` with `<body>`,
- `let Many x = <exp> in let Many x = Many (Many x) in <body>` with `let x = <exp> in <body>`,
- `let Many x = Many <exp> in <body>` with `let x = <exp> in <body>`,
- `let Many f = fix (f, x, .., <exp>, ..) in <body>` with `let rec f x = <exp> in <body>`.

The end result is visible in Figures 3.7 and 3.8: (surprisingly beautiful) OCaml code that is not obviously safe, as promised at the start of the chapter.

It is clear that both Figures 3.3 and 3.8 are realisations of a concise, linearly-typed Kalman filter 3.5 *specification* that describes the intensional behaviour and assumptions (with respect to read/writes permissions, memory management and aliasing) of the program and the BLAS primitives it uses **more accurately** than a Fortran, C or OCaml implementation could.

```
let kalman sigma h mu r_1 data_1 =
  let h, _p_k_n_p_ = Prim.size_mat h in
  let k, n = _p_k_n_p_ in
  let sigma_h = Prim.matrix k n in
  let (sigma, h), sigma_h =
    Prim.symm (Many true) (Many 1.) sigma h (Many 0.) sigma_h
  in
  let (sigma_h, h), r_2 =
    Prim.gemm (Many 1.) (sigma_h, Many false) (h, Many true) (Many 1.) r_1
  in
  let (h, mu), data_2 =
    Prim.gemm (Many 1.) (h, Many false) (mu, Many false) (Many (-1.)) data_1
  in
  let h, new_h = Prim.copy_mat_to h sigma_h in
  let r_2, new_r = Prim.copy_mat r_2 in
  let chol_r, sol_h = Prim.posv new_r new_h in
  let chol_r, sol_data = Prim.potrs chol_r data_2 in
  let () = Prim.free_mat chol_r in
  let h_sol_h = Prim.matrix n n in
  let (h, sol_h), h_sol_h =
    Prim.gemm (Many 1.) (h, Many true) (sol_h, Many false) (Many 0.) h_sol_h
  in
  let () = Prim.free_mat sol_h in
  let h_sol_data = Prim.matrix n (Many 1) in
  let (h, sol_data), h_sol_data =
    Prim.gemm (Many 1.) (h, Many true) (sol_data, Many false) (Many 0.) h_sol_data
  in
  let mu, mu_copy = Prim.copy_mat mu in
  let (sigma, h_sol_data), new_mu =
    Prim.symm (Many false) (Many 1.) sigma h_sol_data (Many 1.) mu_copy
  in
  let () = Prim.free_mat h_sol_data in
  let h_sol_h_sigma = Prim.matrix n n in
  let (sigma, h_sol_h), h_sol_h_sigma =
    Prim.symm (Many true) (Many 1.) sigma h_sol_h (Many 0.) h_sol_h_sigma
  in
  let sigma, sigma_copy = Prim.copy_mat_to sigma h_sol_h in
  let (sigma, h_sol_h_sigma), new_sigma =
    Prim.symm (Many false) (Many (-1.)) sigma h_sol_h_sigma (Many 1.) sigma_copy
  in
  let () = Prim.free_mat h_sol_h_sigma in
  ((sigma, (h, (mu, (r_2, sol_data)))), (new_mu, new_sigma)) )
in
kalman
```

Figure 3.8 – OCaml code for a Kalman filter, generated (at *compile time*) from the code in Figure 3.5, passed through `ocamlformat` for presentation.

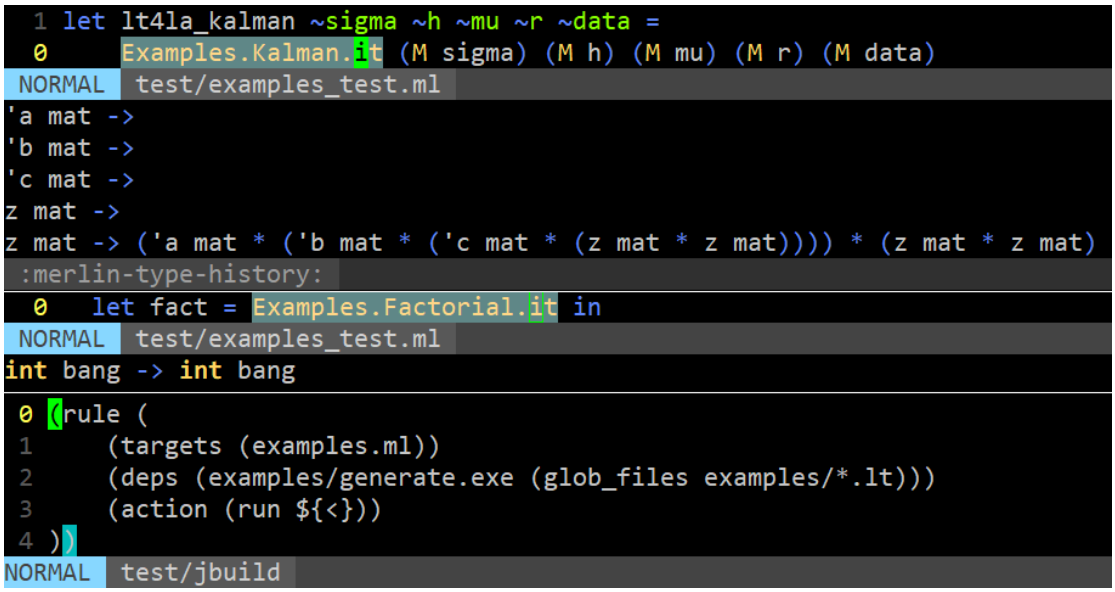

Figure 3.9 – Using LT4LA functions from OCaml with the Dune build system.

3.4.1 Build System

So once you have written your memory-optimised program with all the features and support provided by LT4LA and made it produce well-typed, compilable OCaml code, the question then becomes, how to use this code. This process will vary across ecosystems, but within OCaml, the new build system on the block *Dune* has support for generating, compiling and linking OCaml modules at *compile time* (Figure 3.9). Indeed, this is how I have written tests and benchmarks for programs produced by LT4LA from within OCaml. In particular, I can use my linearly-typed Kalman filter implementation just like and with any other OCaml function.

Generating code at compile time has two advantages. First, it avoids runtime overhead. Second, it catches interface and **relevant implementation changes** between the generated code and the code that uses it. The second advantage is *another benefit* of *embedding* LT4LA's type system inside of OCaml's. Relevant implementation changes are caught because LT4LA's types express some of the intensional behaviour and assumptions of its programs.

I suspect that this approach will be very valuable to not just *users* of libraries such as Numpy or Owl, but also their *implementors*: library functions which present a safe, value-semantic interface but use unsafe, mutating operations on the inside could now be expressed using LT4LA and gain **safety, value-semantics and automatic checking for their implementations**.

3.5 Summary

I explained how a few core features – linearity, the `Many` constructor, value-restriction, fractional-capabilities with inference, if-expressions and recursive functions – are enough to *statically capture and automatically check* aliasing, read/write permissions, memory allocation, re-use and deallocation of non-trivial linear algebra programs. I also demonstrated that simple pattern-matching and desugaring provides the potential for a **new, *type-directed*** approach to matrix expression compilation. Lastly I have shown that it is possible to use these features with *existing* languages and frameworks.[1]

[1] As mentioned in the previous chapter, if the host language supports *syntax-extensions*, like PPX for OCaml, it is possible to construct LT4LA expressions *from within* the host language.

4 Evaluation

I will evaluate the central premise of this thesis: linear types are a practical and usable tool to help programmers write readable (equational and algebraic), safe (with respect to aliasing, read/write permissions, memory re-use and deallocation) and explicit (with respect to memory allocation) code that (1) is more memory-efficient than code written using high-level linear algebra libraries and (2) performs just as predictably as code written using low-level linear algebra libraries. I will also elaborate on the qualitative benefits of using linear types to write linear algebra programs.

4.1 Compared to C

To test whether code written using LT4LA performs just as predictably as code written using low-level linear algebra libraries, I wrote two implementations of a Kalman filter:

1. A CBLAS/LAPACKE implementation, written in C (Figure 4.1), with a minimal number of temporaries, calls to 'symm' for matrices known to be symmetric ahead of time, transposition passed in to 'gemm' as a flag and Cholesky decomposition for multiplying by an inverse of a matrix.

2. A LT4LA implementation (Figure 3.5), also with a minimal number of temporaries, calls to 'symm' for matrices known to be symmetric ahead of time, transposition passed in to 'gemm' as a flag and Cholesky decomposition for multiplying by an inverse of a matrix.

Both implementations produce the same answers (within at most 2^{-52}). Raw output of implementation traces and the benchmarking program (including sample sizes) are in Appendix C.

I compared the implementations on two metrics: memory usage (via number and size of temporaries allocated) and execution time. For the former, I compiled Owl with print-statements on the relevant primitives to see exactly the number of temporaries allocated for a single call of each function. While I did also attempt to use gperftools and OCaml's profiling support with gprof for a more holistic view of memory usage in the presence of OCaml's garbage-collector, I ran into technical difficulties irrelevant to this thesis.

I measured execution time, in micro-seconds, against an exponentially (powers of 5) increasing scaling factor for matrix size parameters $n = 5$ and $k = 3$. For small scaling factors (1, 5, 25), I used the Core_bench micro-benchmarking library, for larger factors (125 and greater), I used the `getrusage` system call (called `Unix.times` in OCaml), sandwiched between calls to `Gc.full_major` to minimise the effects of garbage-collection. Core_bench performs a linear regression (here, time against batch-size) so includes a 95% confidence-interval with R^2. Larger scales have errors reported to $\pm\sigma$ (one standard-deviation).

```
static void kalman( const int n,               const int k,               const double *sigma, /* n,n */
                    const double *h, /* k,n */ const double *mu, /* n,1 */ double *r,           /* k,k */
                    double *data,    /* k,1 */ double **ret_mu,  /* k,1 */ double **ret_sigma   /* n,n */ ) {
        double* k_by_n = (double *) malloc(k * n * sizeof(double));
/*16*/  cblas_dsymm(CblasRowMajor, CblasRight, CblasUpper, k, n, 1., sigma, n, h, n, 0., k_by_n, n);
/*17*/  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, k, k, n, 1., k_by_n, n, h, n, 1., r, k);
/*18*/  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, k, 1, n, 1., h, n, mu, 1, -1., data, 1);
/*19*/  cblas_dcopy(k * n, h, 1, k_by_n, 1);
        double* k_by_k = (double *) malloc(k * k * sizeof(double));
/*20*/  cblas_dcopy(k * k, r, 1, k_by_k, 1);
/*21*/  LAPACKE_dposv(LAPACK_ROW_MAJOR, 'U', k, n, k_by_k, k, k_by_n, n);
/*23*/  LAPACKE_dpotrs(LAPACK_ROW_MAJOR, 'U', k, 1, k_by_k, k, data, 1);
        free(k_by_k);
        double* n_by_n = (double *) malloc(n * n * sizeof(double));
/*24*/  cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n, n, k, 1., h, n, k_by_n, n, 0., n_by_n, n);
        free(k_by_n);
        double* n_by_1 = (double *) malloc(n * sizeof(double));
/*25*/  cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n, 1, k, 1., h, n, data, 1, 0., n_by_1, 1);
        double* new_mu = (double *) malloc(n * sizeof(double));
/*26*/  cblas_dcopy(n, mu, 1, new_mu, 1);
/*27*/  cblas_dsymm(CblasRowMajor, CblasLeft, CblasUpper, n, 1, 1., sigma, n, n_by_1, 1, 1., new_mu, 1);
        free(n_by_1);
        double* n_by_n2 = (double *) malloc(n * n * sizeof(double));
/*28*/  cblas_dsymm(CblasRowMajor, CblasRight, CblasUpper, n, n, 1., sigma, n, n_by_n, n, 0., n_by_n2, n);
/*29*/  cblas_dcopy(n*n, sigma, 1, n_by_n, 1);
/*30*/  cblas_dsymm(CblasRowMajor, CblasLeft, CblasUpper, n, n, -1., sigma, n, n_by_n2, n, 1., n_by_n, n);
        free(n_by_n2);
        *ret_sigma = n_by_n;
        *ret_mu = new_mu; }
```

Figure 4.1 – CBLAS/LAPACKE implementation of a Kalman filter. I used C instead of Fortran because it is what Owl uses under the hood and OCaml FFI support for C is better and easier to use than that for Fortran. A distinct 'measure_kalman' function that sandwiches a call to this function with `getrusage` is omitted for brevity.

4.1.1 Memory Usage

Inspecting the trace for LT4LA shows 6 calls to 'empty' (function for allocating new matrices): 4 temporaries plus 2 for storing the results. From the CBLAS implementation (Figure 4.1), we can see that it too has 6 calls to malloc: 4 temporaries plus 2 for storing the results.

4.1.2 Execution Time

A graph of the execution times (with error bars which are present but quite small) is shown in Figure 4.2.

For $n = 5$, CBLAS was faster ($24\mu s$, 526 samples) than LT4LA ($41\mu s$, 466 samples). For $n = 25$, and around 350 samples each, CBLAS was again faster ($104\mu s$) than LT4LA ($133\mu s$). The 95% confidence-intervals for these measurements differ from the means by at most $2\mu s$. For $n = 125$, and around 110 samples each, CBLAS was *slower* ($1803\mu s$ [1746, 1867]) than LT4LA ($1678\mu s$ [1646, 1714]). For $n = 625$ and 1000 samples each, they took a *very similar* amount of time: LT4LA took $180.5 \pm 38\,ms$ and CBLAS took $188 \pm 36\,ms$. Despite the large sample size, the standard-deviation is still quite high; however, *because* of the large sample size, the p-value (Welch's t-test) is very small ($p < .05$), suggesting that the difference in the means is statistically significant. Lastly, for $n = 3125$ and 15 samples each, LT4LA and CBLAS continued to take a similar amount of time ($16.1 \pm 0.19\,s$ and $15.7 \pm 0.53\,s$ respectively, $p < .05$ using Welch's t-test).

4.1.3 Analysis

Having access to primitives which allow a programmer to re-use memory means that memory usage for temporaries in LT4LA is on par with that of CBLAS. One caveat is that the `freeM` primitive is a no-op in LT4LA, so deallocation still relies on OCaml's garbage-collector.

For small matrix sizes, LT4LA and CBLAS execution times differ. I suspect this is due to large sample sizes causing more allocations and thus more garbage-collector unpredictability (which the linear regression *did not* take into account; multi-variate regressions are not fully supported by Core_bench yet). As matrix size increases, execution times of LT4LA become very similar to those of CBLAS.

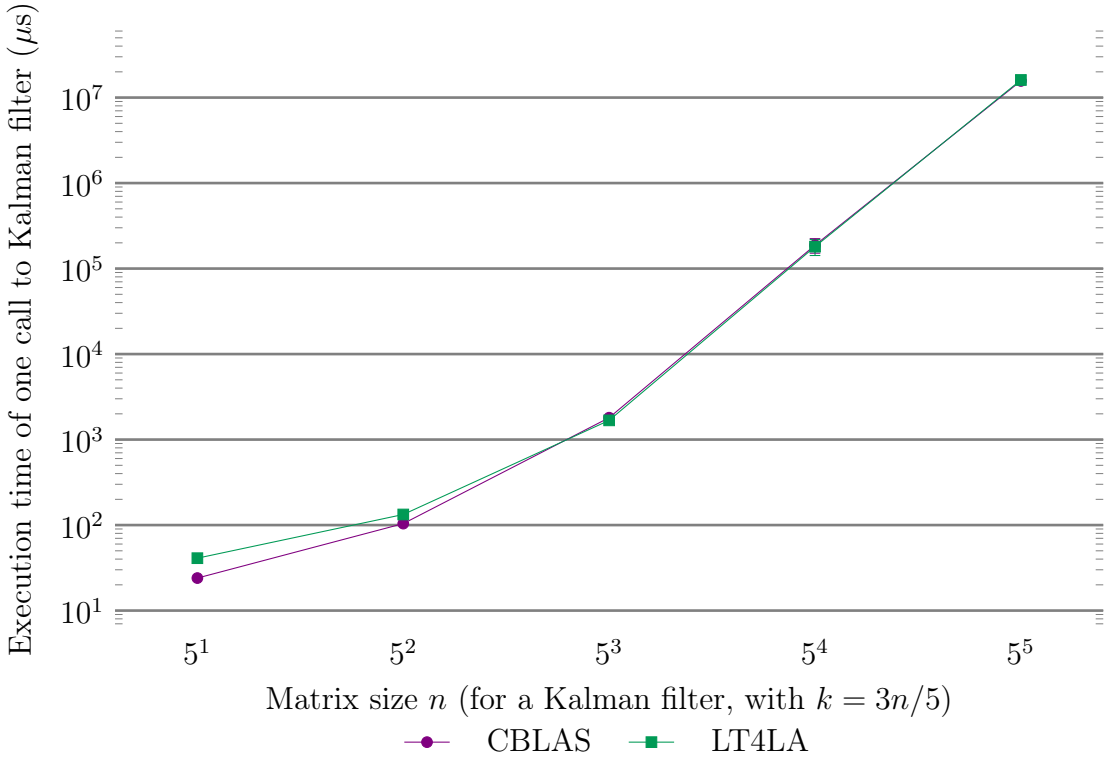

Figure 4.2 – Comparison of execution times (error bars are present but quite small). Small matrices and timings $n \leq 5^3$ were micro-benchmarked with the Core_bench library. Larger ones used Unix's `getrusage` functionality, sandwiched between calls to `Gc.full_major` for the OCaml implementations.

Therefore, by using LT4LA, it is possible to write linear algebra programs that perform just as predictably (with regards to memory usage and execution time) as code written using low-level libraries, but gain readability (equational and algebraic expressions) and safety (with respect to aliasing and read/write permissions).

4.2 Compared to OCaml

To test whether code written using LT4LA is more memory-efficient than code written using high-level libraries, I wrote two[1] more implementations of a Kalman filter:

1. An Owl/OCaml implementation using a Cholesky decomposition (Figure 4.3) but not taking advantage of matrices known to be symmetric ahead

[1]I attempted a third implementation using Owl's *Lazy* module but it did not support matrix inversion at the time of writing.

```
let potrs ~uplo a b =
  let b = Owl.Mat.copy b in
  Owl.Lapacke.potrs ~uplo ~a ~b
;;

let chol_kalman ~sigma ~h ~mu ~r ~data =
  let open Owl.Mat in
  let ( * ) = dot in
  let h' = transpose h in
  let sigma_h' = sigma * h' in
  let chol = Owl.Linalg.D.chol (r + h * sigma_h') in
  let sigma_h'_inv rest = sigma_h' * potrs ~uplo:'U' chol rest in
  let new_sigma = sigma - sigma_h'_inv (h * sigma) in
  let new_mu = mu + sigma_h'_inv (h * mu - data) in
  ((sigma, (h, (mu, (r, data)))), (new_mu, new_sigma))
;;

let owl_kalman ~sigma ~h ~mu ~r ~data =
  let open Owl.Mat in
  let ( * ) = dot in
  let h' = transpose h in
  let sigma_h' = sigma * h' in
  let x = sigma_h' * (inv @@ r + h * sigma_h') in
  let new_mu = mu + x * (h * mu - data) in
  let new_sigma = sigma - x * h * sigma in
  ((sigma, (h, (mu, (r, data)))), (new_mu, new_sigma))
;;
```

Figure 4.3 – Implementations of a Kalman filter using Owl, top one using a Cholesky decomposition, bottom one using idiomatic Owl. Owl does not yet provide a non-mutating 'potrs' function, so I wrote my own which returns a mutated *copy* of its argument instead.

of time, and producing a new temporary matrix for every operation (including inverse and transpose).

2. An idiomatic Owl/OCaml implementation (Figure 4.3) with an explicit inverse (LU decomposition), not taking advantage of matrices known to be symmetric ahead of time, and producing a new temporary matrix for every operation (including inverse and transpose).

These implementations also produce the same answers (within at most 2^{-52}) as their LT4LA and CBLAS counterparts and their data is also included in Appendix C.

4.2.1 Memory Usage

Inspecting the Owl trace shows it used 11 temporary matrices (13 calls to empty, 2 of which are the resulting matrices). The same for Chol shows it used 13 temporaries (same as Owl plus two temporaries for the two calls to potrs). Analysing the sub-expressions of the Owl implementation shows the total amount of memory allocated for temporaries is $n + n^2 + 4nk + 3k^2 + 2k$ words; for Chol the total is that of Owl plus $n + nk$.

4.2.2 Analysis

For LT4LA and CBLAS the total amount of memory allocated for temporaries is $n + n^2 + nk + k^2$. The difference between these two implementations and the idiomatic Owl implementation is $k(3n + 2k + 2)$ words. Hence, by using LT4LA, it is possible to have the readability (equational and algebraic expressions) and safety of high-level libraries, and gain precise and explicit control over memory allocation and re-use.

4.3 Limitations

I chose the example of a Kalman filter because it is used in the real world, consists purely of a sequence of matrix expressions and produces many unnecessary temporary matrices when implemented idiomatically in a high-level library. It is good for isolating the key differences between not having and having linear types to help a programmer safely manage memory, aliasing, and read/write permissions, whilst excluding other aspects also important to real world linear algebra programs such as control flow or blocking.

4.3.1 Curious Behaviour

A graph of the execution times (with error bars which are present but quite small) of *all four* implementations is show in Figure 4.4.

For $n = 25$ and $n = 125$, the idiomatic Owl implementation is the *fastest* of them all at $95\mu s$ and $1488\mu s\,[1464, 1515]$ respectively. But then for $n = 625$ and $n = 3125$, Chol is the fastest, at $125.5 \pm 25\,ms$ and $11.2 \pm 0.85\,s$ respectively

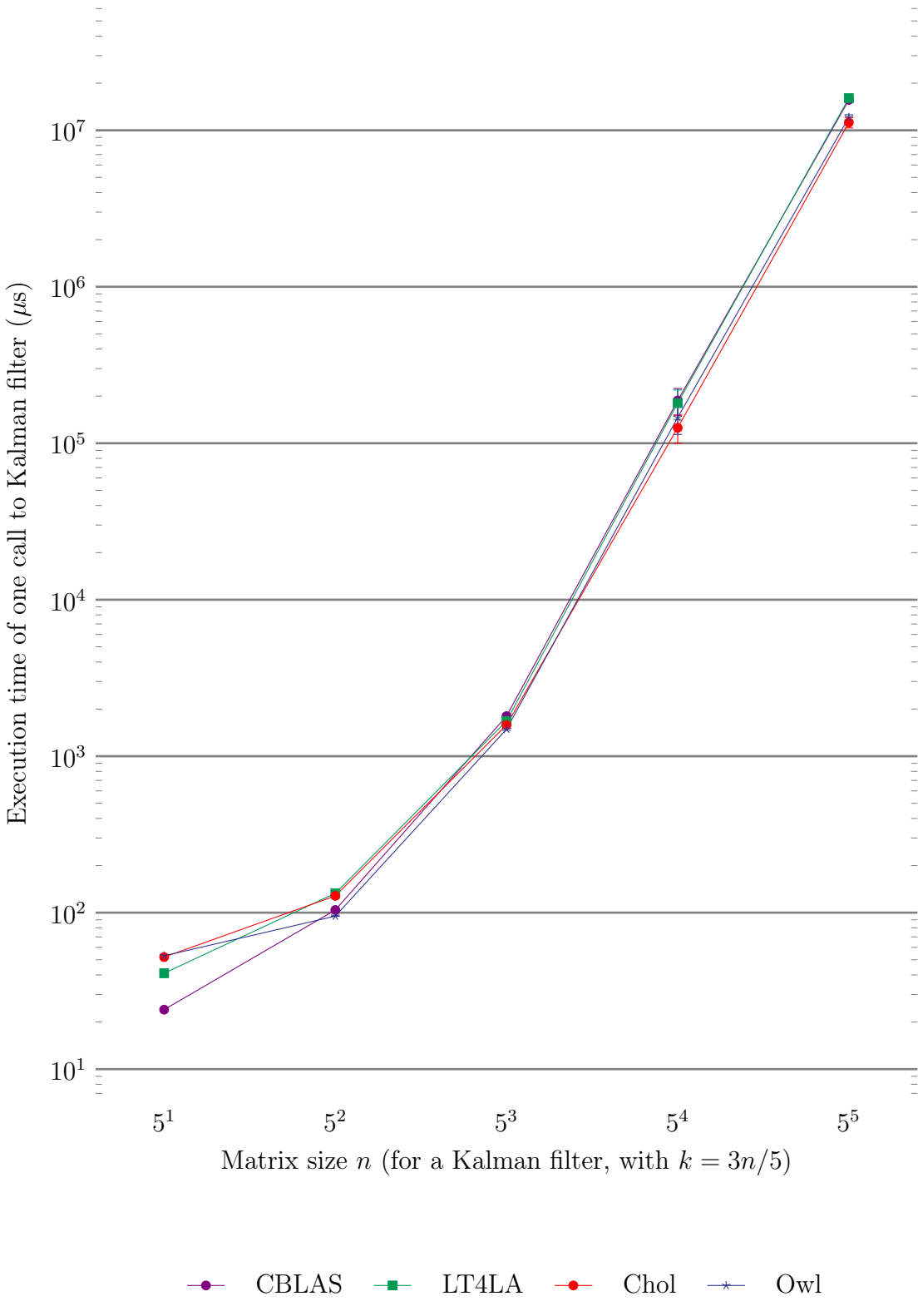

Figure 4.4 – Comparison of execution times (error bars are present but quite small). Small matrices and timings $n \leq 5^3$ were micro-benchmarked with the Core_bench library. Larger ones used Unix's `getrusage` functionality, sandwiched between calls to `Gc.full_major` for the OCaml implementations.

(with Owl a close second for the latter at $12.1 \pm 0.47\,s$, a statistically significant difference with $p < .05$ using Welch's t-test).

The trend here is that for $n = 625$ and $n = 3125$ (sizes for which `Gc.full_major` was called before each measurement), mean execution times start to split into two groups: Chol and Owl as one group (similar, *faster* times) and LT4LA and CBLAS as another (similar, *slower* times). *Although the goals of this thesis have been met*, this is unexpected behaviour, and attempting to understand *why* the Chol/Owl implementations tended to be faster for anything except the smallest of matrices, baffled several people I consulted, including experts at OCamlLabs.

Here are several reasons why this behaviour is surprising:

- More temporaries means more garbage-collection pressure meaning more frequent garbage collection, especially as matrix sizes grow.
- CBLAS/LT4LA use less memory so in principle should have a smaller working set and better temporal and spatial locality for cache.
- CBLAS/LT4LA use routines that combine multiplication and addition of matrices (such as gemm, and symm).
- CBLAS/LT4LA have *four* calls to the 'symm' routine, which performs *half* as many multiplications as 'gemm' under the assumption that one of its arguments is a symmetric matrix.
- CBLAS/LT4LA use Cholesky decomposition: it performs *half* as many operations as LU decomposition (as used by Owl) under the assumption that its argument is a symmetric matrix.
- All of the implementations either directly or indirectly (via Owl's bindings) call the same set of CBLAS/LAPACKE bindings.

Here are things I checked:

- **Cache miss-rates.** Running the different implementations through the Cachegrind cache simulator (part of Valgrind) showed that LT4LA/CBLAS had a roughly 1% higher cache miss rate than Chol/Owl (rising from around 11% to around 12%). This seemed insufficient to account for the differences, so I investigated further.
- **Data access patterns.** I added an extra, modified implementation of LT4LA, which transposed the 'h' parameter into a new matrix rather than

using the transpose flag to 'gemm', to see if row-vs-column access patterns could account for the differences. They did not.

Had I been able to use gprof or gperftools, I would have profiled the remaining key difference: the 'symm' routine, just to be able to eliminate it as a suspect.

A **positive take-away** from all of this is that LT4LA at least gives a programmer *choice* and *control* about how to optimise their program to their needs because (1) it met its goal of enabling a programmer to write code that performs just as predictably as code written using low-level libraries (2) it does so *without* the associated dangers of unenforced and uncheckable rules regarding aliasing, read/write permissions, memory allocation, re-use and deallocation that would come with using C, Fortran, or the unsafe primitives of Owl.

4.4 Qualitative Benefits

We have already seen (last chapter and this chapter) that linear types help programmers write readable, safe (with respect to aliasing, read/write permissions, memory re-use and deallocation) and explicit (with respect to memory allocation) code. To justify their why I think they are a *practical* and *usable* way to do so, I will elaborate on the qualitative benefits I experienced whilst using LT4LA.

Prior to this project, I had no experience with linear algebra libraries or the problem of matrix expression compilation. As such, I based my initial LT4LA implementation of a Kalman filter using BLAS and LAPACK, on a popular GitHub gist of a Fortran implementation, one that was automatically generated from SymPy's matrix expression compiler [8].

Once I translated the implementation from Fortran to LT4LA, I attempted to compile it and found that (to my surprise) it did not type-check. This was because the original implementation contained incorrect aliasing, unused and unnecessary temporaries, and did not adhere to Fortran's read/write permissions (with respect to `intent` annotations `in`, `out` and `inout`) all of which were now highlighted by LT4LA's type system.

The original implementation used 6 temporaries, one of which was immediately spotted as never being used due to linearity. It also contained two variables which were marked as `intent(in)` but would have been written over by calls to 'gemm', spotted by the fractional-capabilities feature. Furthermore, it used a matrix *twice*

in a call to 'symm', once with a read permission but once with a *write* permission. Fortran assumes that any parameter being written to is not aliased and so this call was not only incorrect, but illegal according to the standard, both aspects of which were captured by linearity and fractional-capabilities. Lastly, it contained another unnecessary temporary, however one that was not obvious without linear types. To spot it, I first performed live-range splitting (checked by linearity) by hoisting calls to `freeM` and then annotated the freed matrices with their dimensions. After doing so and spotting two disjoint live-ranges of the same size, I replaced a call to `freeM` followed by allocating call to `copy` with one, in-place call to `copyM_to`. I believe the ability to boldly refactor code which manages memory is good evidence of the usefulness of linearity as a tool for programming.

4.5 Summary

Writing a linear algebra program using LT4LA combines the best of low- and high-level libraries: it gives a programmer *explicit control* (over read/write permissions, aliasing and memory allocation, re-use and deallocation), readability (equational and algebraic expressions) *and* safety (automatic checking). Programs written using it *signal precise intent* to the reader and compiler and perform similarly to equivalent programs written directly using low-level libraries. In turn, such programs can be *checked automatically* against their intent, especially when *refactoring* or *rewriting* code. Although any expert *could* have followed the same line of reasoning laid out above, and arrived at the same program, LT4LA's type system enables a non-expert (yours truly) to do the same with increased confidence in the result[2] by checking said reasoning.

[2]This does not preclude testing code by actually *executing* it, but definitely *complements* it.

5 | Related Work

Now that I have described my contributions, I will explain how it relates to existing work, leaving brief discussions on future work to the next chapter. I strongly believe the field of matrix expression compilation would benefit greatly from a comprehensive literature review but unfortunately that is beyond the scope of this chapter.

5.1 Matrix Expression Compilation

Most of the projects below try to be fully-automated black-boxes which model computing a matrix expression as some sort of graph with informal, ad-hoc rules about what can and should be copied or modified in-place. Allocations, temporaries and common sub-expressions are details invisible to the programmer, left to the compiler.

The matrix expression 'compiler' as implemented in LT4LA is intended to be a mere proof-of-concept of how linear types can and arguably should be used *ergonomically*. I have taken the approach of attempting to help programmers precisely and explicitly capture, using types, the practices prevalent in code they already write.

I believe the advantages of my approach are two-fold (1) *predictable* performance (as defined in Chapter 4) and (2) more accurate modelling of how low-level kernels handle their resources. My confidence in the latter claim comes from finding errors in Fortran code output by SymPy to compute a Kalman filter (as described in Section 4.4); errors that would not have passed type-checking had it been translated via LT4LA as an intermediate representation.

5.1.1 SymPy

SymPy is a symbolic computer algebra system for Python; its matrix expression compiler [8] uses a term-rewrite system, with rules supplied by a BLAS expert (which must be strongly-normalising, that is, never cause a loop) but need not be confluent (there can be more than one solution per expression). Rules include expressions to match on, the expression it can produce, information about the expressions (such as whether the matrix is symmetric or full-rank) and information about which variable is updated in-place.

5.1.2 Clak and Cl1ck

Clak and Cl1ck [9] were developed independently around the same time as SymPy's matrix expression compilation. Clak attempts to produce *multiple* algorithms for a single matrix expression, by considering a wider matrix expression grammar and more matrix properties and inference rules. These algorithms assume basic

building blocks such as products and factorisations. Cl1ck attempts to take on the challenge of writing BLAS/LAPACK-like libraries too, by generating low-level loop-based blocked routines for the aforementioned basic building blocks, in the spirit of the FLAME [10] project.

5.1.3 Linnea and Taco

Linnea [11] and Taco [12] are two newer contenders to Clak and Cl1ck respectively. Linnea continues the work of Clak to producing real executable code for *existing* libraries and kernels, as well as incorporating work on a *generalised* matrix chain algorithm [13]. Taco (*Tensor Algebra COmpiler*) focuses on emitting efficient routines for expressions in tensor index notation, with many optimisations for *sparse* tensors.

5.2 Metaprogramming

Most of the compilers in the aforementioned projects usually built, analysed, compiled, ran (and in some cases, dynamically loaded) expressions (including functions) at runtime, similar to how regular expressions are handled in most languages – in particular, even when the regular expression is known at compile time.

In LT4LA, I took the approach of having a concrete syntax and expression language which was then translated to OCaml and made available to other modules *at compile time* via the build system. There was nothing inherent in the approach that prevented me from using OCaml's PPX syntax-extensions so that I could write normal OCaml expressions from within OCaml and have them checked for linearity before compilation.

Having a statically compiled language and a build system as so affords the advantage of eliminating the runtime overheads mentioned at the start of this section. However, there is some useful information (such as matrix dimensions for the matrix chain algorithm) which is sometimes known only at runtime (but once known, usually fixed). In these cases, using *multi-stage programming* would be a better approach to implementing a matrix expression compiler.

5.2.1 MetaOCaml and Scala LMS

MetaOCaml [14] and Scala with Lightweight Modular Staging [15] are systems which support multi-stage programming. A typical example of this is the generation of Fast Fourier-Transform kernels, specialised to a desired array length. Combining this with recent work on generalising and automatically deriving *partially static* representations of data [16], it may be possible to apply such techniques to *tensor algebra* expressions.

5.2.2 Expression Templates in C++ Libraries

Expression templates are a commonly used compile-time metaprogramming technique, used by Eigen, uBLAS and Armadillo to name a few. If known at compile-time, matrix dimensions can also be passed in as template arguments to ensure operations match (otherwise checking at runtime). In Eigen, such features are combined with heuristics to enable *lazy evaluation* and automatically determine whether a sub-expression is evaluated into a temporary variable or not.

Libraries that use expression templates usually perform rudimentary pattern-matching and in some cases, loop-fusion, to avoid evaluating expressions in a purely binary manner (preventing the bane of a C++ programmer: temporaries and unnecessary copies) when possible (either by translating to a library kernel call or, as an example, inlining a $v := a + b + c$ vector expression into one loop).

Therefore, expression templates eliminate some runtime overheads, but not all (unlike LT4LA), because the heuristics they use rely on statically imperfect information. They also do not allow a user to easily inspect the generated code (also unlike LT4LA), thus losing explicitness.

5.3 Types

Apart from lazy evaluation, the following projects show how instead of a (E)DSL library approach, we could have type-level resource management provided far more conveniently and naturally at the *language* level. The difference is that a library can be designed, shipped and used now whereas language features take time and can have unintended interactions with other language features. My hope is that

once people are convinced of the utility type-level resource management by using a library, the impetus for integrating such features into a language follows.

5.3.1 Lazy Evaluation

A particularly strong advantage LT4LA has over other libraries that use lazy evaluation is, funnily enough, linear types; more precisely, it is the static and perfect information they guarantee about aliasing: with Owl, this would be comparable to assuming every vertex of its computation graph has only one edge in and out; with Eigen, this would be comparable to having a `noalias` annotation on every "assignment", **without the danger of getting it wrong** and invoking undefined behaviour.

This simplifies the rules and exceptions a programmer reasoning about memory usage needs to remember. Of course, now the programmer has to figure out how they are using their temporaries, but because matrices are linearly typed, redundant copies and missed frees can be pointed out by the compiler, thus guiding the programmer towards a satisfying solution.

5.3.2 Futhark

Futhark [17] is a second-order (meaning it supports functions such as map and fold/reduce) array combinator (meaning array operations can be fused into streams to reduce temporaries) language designed for efficient parallel compilation. It supports ML-style modules, loops, limited parametric polymorphism, size types and uniqueness types (an idea closely related to linear types). Its combinators are more expressive than typical linear algebra library kernels; hence it encourages shorter, more declarative code.

5.3.3 Substructural Features in Rust

Rust [18] is a (relatively) new systems programming language aiming to bring the last two decades of programming language research to the masses in a usable and friendly manner. Its *borrow checker* is the feature most relevant to this thesis because it also statically attempts to prevent many resource-related bugs. There are a few linear algebra libraries for Rust under development; careful use of its macro system and borrow checker could make it the safest and easiest to use

language for linear algebra projects to come. Its struggle is more likely to be against the inertia of the large amounts of C, C++ and Fortran code already out there rather than its usability or benefits. Given this inertia, I believe there is value in taking the DSL library approach of LT4LA, which will work with existing systems.

5.3.4 Linear Types in Haskell

Linear types have been incorporated into a branch of the Glasgow Haskell Compiler [19] in an attempt to provide safe, functional streaming and IO (after people saw the potential from libraries providing linearity features). Practical benefits include zero-copy buffers and eliminating garbage collection in certain situations by allowing the user to safely manage memory. The fact that it can and has been done gives me hope that other languages will also see the value and adopt some form of resource-management in their type systems.

5.3.5 Linear and Dependent Types in Idris

Dimension mismatches are seen as an irritating but small inconvenience when writing linear algebra code. However, another error I found in the Fortran code output by SymPy to compute a Kalman filter was a dimension/transposition error. Although we would not need full dependent types to solve dimension mismatches (symbolic size types would be sufficient), managing properties about matrices could be done at the type-level in a dependently typed setting.

We could then express the usual properties and results of operations at the type-level, ensuring, for example, that certain functions are called only when the matrix is symmetric and can be written to. Idris (a Haskell inspired language with dependent types) has had experimental support for uniqueness types since its early days and now a linear types extension [20] is also being worked on based on new research around integrating linear and dependent types [5].

6 | Conclusion

In this thesis, I presented **linear types with fractional-capabilities** as a type-based formalism for expressing **aliasing, read/write permissions, memory allocation, re-use and deallocation**; I provided a detailed description of each feature of the type system, its intent and implementation in Section 3.2.

I used that formalism to make **precise and explicit the intensional behaviour and assumptions** (about aliasing, read/write permissions and memory re-use) of linear-algebra library APIs, first with the simplified example of the fictional 'simple_dgemm' primitive in Section 2.3 – where I showed how **the types made it impossible to use 'simple_dgemm' incorrectly** – and then, more realistically with the types of LT4LA primitives (including actual CBLAS/LAPACKE routines) listed in Appendix B.

I also used that formalism to implement a simple, yet effective matrix expression 'compiler' based on pattern-matching. I was able to use to that compiler to write, a **non-trivial, yet readable** (equational and algebraic) linear algebra program (Kalman filter, as described in Section 3.3). I shared how, in doing so, I could use the type-checking as a tool to **automatically highlight bugs** regarding aliasing, read/write permissions and memory re-use of a reference implementation (Section 4.4) and the matrix expression compiler that generated it (Section 5.1).

I showed that this implementation was **more memory-efficient** than one written using a high-level library and performed just as **predictably** as one written using a low-level library, enabling a programmer to have the best of both worlds (Chapter 4). The type system ensured that the implementation was **safe and explicit** with respect to aliasing, read/write permissions, memory allocation, re-use and deallocation.

Lastly, I was able to provide all of this functionality as a **usable OCaml library**, by generating **readable and not obviously safe** OCaml code (1) which looks like it is written by an expert and (2) whose types reflect its intensional behaviour and

assumptions (interpreting types into OCaml, Figure 3.1). This library, LT4LA, produces **helpful type-errors** (Section 2.3), comes with an interactive REPL and is documented and well-tested (Section 3.1).

6.1 Future Work

LT4LA is a proof-of-concept design for linearly typed DSL for writing linear-algebra programs in a way that can work with existing languages and libraries. It is made of up of the core language and its matrix expression 'compiler'; both could be extended in many ways.

6.1.1 Core Language

Although I use well-established type system features, it may be worthwhile to formally state and verify my claimed safety properties. The logical conclusion of LT4LA's core language would be a language expressive enough to provide a usable, linearly typed interface to the unsafe parts of the Owl library for OCaml, perhaps as a functor like its current, 'Lazy' interface. To do that, the language implementation would need to support more than just 64-bit floating-point numbers for elements (for example, 16-bit, 32-bit or complex), polymorphism, a unified way of dealing with arrays and matrices (perhaps through sub-typing). It may also be useful to explore localised linearity *inference*, to automate the reasoning I did manually when implementing a memory-efficient Kalman filter (Section 4.4).

Another, equally valid but opposing direction may be to decrease the amount of expressivity, to bring it closer to Fortran and C so that it would be easier to emit code in those languages. A full set of linearly typed BLAS/LAPACK bindings (to encode information in documentation at the type level) would be a good first step, perhaps leading into more research on formal semantics for Fortran.

6.1.2 Matrix Expression Compilation

All of the matrix expression compilers mentioned in the previous chapter construct some sort of data-flow graph to represent the computation being executed. While this seems intuitive, there is no formal argument for this approach. Some directions

in which a type-based approach to efficient matrix expression compilation could be taken are:

- **As a typed IR** for matrix expression compilers. This in turn could enable
  - existing matrix expression compilers to be less opaque about what resources they are consuming.
  - open up opportunities for non-local sharing of temporary values with some intra-procedural analysis.
  - allow the user to choose: use a matrix expression compiler when desired and drop down to a usable typed IR for finer control, whilst still retaining safety guarantees.
- **Formal verification of matrix expression compilers** by precisely specifying the intensional and extensional properties of the source and target languages.
- **Multi-stage programming** to use information only available at runtime (such as sizes, matrix properties, sparsity, control flow) for generating more specialised code.
- **Dependent types** to have control over how resources can be used and split. In addition to formal verification, dependent types could be combined with linearity to express finer-grain conventions surrounding blocking, slicing and writing to *parts* of the matrix instead of the whole. This is already prevalent with 'dsymm' like BLAS routines which only read the lower or upper triangle of a matrix. This idea is inspired by Conor McBride's talk on writing to terminals with "Space Monads" [21].
- **Compiling to hardware** is also an option - once we know exactly when and where temporaries are required and what can be re-used when, we come one (small, but useful) step closer to realising matrix expressions directly on hardware.

A | Ott Specification

The following pages present a specification of the grammar and type system used in LT4LA, produced using the Ott [22] tool. It is important to note that the type system is not implemented how it is described in the coming pages. For explaining (and using Ott) it was easier to set it out as below. However, for implementing, I found it much more and user- and debugging-friendly to:

- Have the type environment *change* as a result of type-checking an expression, similar to the rules shown in Figure 2.1; with this, the below rules describe the *difference* between the environment after and before checking an expression. For example, in the pair-introduction rule, $\Gamma = \Gamma_2 - \Gamma_1$ and $\Gamma' = \Gamma_3 - \Gamma_2$, for an appropriate definition of $(-)$.
- *Mark* variables as used instead of *removing* them from the environment for better error messages.
- Have *one* environment where variables were *tagged* as linear and unused, linear and used, and intuitionistic. This was definitely an implementation convenience so that variable binding could be handled uniformly for linear and intuitionistic variables and scoping/variable look-up could be handled implicitly with the associative-list structure of the environment. So, it would be more (but still not completely) accurate to define the variable rule as:

$$\frac{}{\Theta; \Gamma, x \overset{n}{:} t \vdash x : t; \Gamma, x \overset{n-1}{:} t} \textsc{Ty\_Var}$$

for $n \in \{0\,(\text{used}), 1\,(\text{unused}), \omega\,(\text{intuitionistic})\}$, $\omega - 1 = \omega$ and $1 - 1 = 0$.

fc	fractional capability variable		
x, g, a, b	expression variable		
k	integer variable		
el	array-element variable		
$symb$	::=		
	| λ		
	| $\otimes$		
	| $\multimap$		
	| $\vdash$		
	| $\in$		
	| $\forall$		
	| Cap		
	| Type		
	| !		
	| $\rightarrow$		
	| **value**		

f	::=			fractional capability
	|	fc		variable
	|	**Z**		zero
	|	**S** f		successor

t	::=			linear type
	|	**unit**		unit
	|	**bool**		boolean (true/false)
	|	**int**		63-bit integers
	|	**elt**		array element
	|	f **arr**		arrays
	|	f **mat**		matrices
	|	$!t$		multiple-use type
	|	$\forall\, fc.t$	bind fc in t	frac. cap. generalisation
	|	$t \otimes t'$		pair
	|	$t \multimap t'$		linear function
	|	$t\{f/fc\}$	M	substitution
	|	(t)	S	parentheses

e	::=			expression
	|	p		primitives
	|	x		variable
	|	**let** $x = e$ **in** e'	bind x in e'	let binding
	|	$()$		unit introduction
	|	**let** $() = e$ **in** e'		unit elimination
	|	**true**		true
	|	**false**		false
	|	**if** e **then** e_1 **else** e_2		if
	|	k		integer

	el		array element
	Many e		!-introduction
	let Many $x = e$ **in** e'		!-elimination
	fun $fc \to e$		frac. cap. abstraction
	$e[f]$		frac. cap. specialisation
	(e, e')		pair introduction
	let $(a, b) = e$ **in** e'	bind $a \cup b$ in e'	pair elimination
	fun $x : t \to e$	bind x in e	abstraction
	$e\ e'$		application
	fix $(g, x : t, e : t')$	bind $g \cup x$ in e	fixpoint

p	::=		primitive
	set		array index assignment
	get		array indexing
	$(+)$		integer addition
	$(-)$		integer subtraction
	$(*)$		integer multiplication
	$(/)$		integer division
	$(=)$		integer equality
	$(<)$		integer less-than
	$(+.)$		element addition
	$(-.)$		element subtraction
	$(*.)$		element multiplication
	$(/.)$		element division
	$(= .)$		element equality
	$(< .)$		element less-than
	not		boolean negation
	share		share array
	unshare		unshare array
	free		free arrary
	array		Owl: make array
	copy		Owl: copy array
	sin		Owl: map sine over array
	hypot		Owl: $x_i := \sqrt{x_i^2 + y_i^2}$
	asum		BLAS: $\sum_i \lvert x_i \rvert$
	axpy		BLAS: $x := \alpha x + y$
	dot		BLAS: $x \cdot y$
	rotmg		BLAS: see its docs
	scal		BLAS: $x := \alpha x$
	amax		BLAS: $\operatorname{argmax} i : x_i$
	setM		matrix index assignment
	getM		matrix indexing

	shareM	share matrix
	unshareM	unshare matrix
	freeM	free matrix
	matrix	Owl: make matrix
	copyM	Owl: copy matrix
	copyM_to	Owl: copy matrix onto another
	gemm	BLAS: $C := \alpha A^{T?} B^{T?} + \beta C$
	symm	BLAS: $C := \alpha AB + \beta C$
	posv	BLAS: Cholesky decomp. and solve
	potrs	BLAS: solve with given Cholesky

Θ	::=		fractional capability environment
	|	.	
	|	Θ, fc	

Γ	::=		linear types environment
	|	.	
	|	$\Gamma, x : t$	
	|	Γ, Γ'	

Δ	::=		linear types environment
	|	.	
	|	$\Delta, x : t$	

formula	::=		
	|	*judgement*	
	|	$x : t \in \Delta$	
	|	$x : t \in \Gamma$	
	|	$fc \in \Theta$	
	|	**value** (e)	

Well_Formed	::=		
	|	$\Theta \vdash f$ Cap	Valid fractional capabilities
	|	$\Theta \vdash t$ Type	Valid types

Values	::=		
	|	**value** (e)	Value restriction for !-introduction

Types	::=		
	|	$\Theta; \Delta; \Gamma \vdash e : t$	Typing rules for expressions

judgement	::=	
	|	*Well_Formed*

$$
\begin{array}{lll}
 & | & \mathit{Values} \\
 & | & \mathit{Types}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{user\_syntax} & ::= & \\
 & | & \mathit{fc} \\
 & | & x \\
 & | & k \\
 & | & \mathit{el} \\
 & | & \mathit{symb} \\
 & | & f \\
 & | & t \\
 & | & e \\
 & | & p \\
 & | & \Theta \\
 & | & \Gamma \\
 & | & \Delta \\
 & | & \mathit{formula}
\end{array}
$$

$\boxed{\Theta \vdash f\ \mathsf{Cap}}$ Valid fractional capabilities

$$\frac{\mathit{fc} \in \Theta}{\Theta \vdash \mathit{fc}\ \mathsf{Cap}}\ \textsc{WF\_Cap\_Var}$$

$$\frac{}{\Theta \vdash \mathbf{Z}\ \mathsf{Cap}}\ \textsc{WF\_Cap\_Zero}$$

$$\frac{\Theta \vdash f\ \mathsf{Cap}}{\Theta \vdash \mathbf{S}\ f\ \mathsf{Cap}}\ \textsc{WF\_Cap\_Succ}$$

$\boxed{\Theta \vdash t\ \mathsf{Type}}$ Valid types

$$\frac{}{\Theta \vdash \mathbf{unit}\ \mathsf{Type}}\ \textsc{WF\_Type\_Unit}$$

$$\frac{}{\Theta \vdash \mathbf{bool}\ \mathsf{Type}}\ \textsc{WF\_Type\_Bool}$$

$$\frac{}{\Theta \vdash \mathbf{int}\ \mathsf{Type}}\ \textsc{WF\_Type\_Int}$$

$$\frac{}{\Theta \vdash \mathbf{elt}\ \mathsf{Type}}\ \textsc{WF\_Type\_Elt}$$

$$\frac{\Theta \vdash f\ \mathsf{Cap}}{\Theta \vdash f\ \mathbf{arr}\ \mathsf{Type}}\ \textsc{WF\_Type\_Array}$$

$$\frac{\Theta \vdash t\ \mathsf{Type}}{\Theta \vdash !t\ \mathsf{Type}}\ \textsc{WF\_Type\_Bang}$$

$$\frac{\Theta, \mathit{fc} \vdash t\ \mathsf{Type}}{\Theta \vdash \forall\, \mathit{fc}.t\ \mathsf{Type}}\ \textsc{WF\_Type\_Gen}$$

$$\frac{\Theta \vdash t\ \mathsf{Type} \quad \Theta \vdash t'\ \mathsf{Type}}{\Theta \vdash t \otimes t'\ \mathsf{Type}}\ \textsc{WF\_Type\_Pair}$$

$$\frac{\Theta \vdash t\ \mathsf{Type} \quad \Theta \vdash t'\ \mathsf{Type}}{\Theta \vdash t \multimap t'\ \mathsf{Type}}\ \textsc{WF\_Type\_Lolly}$$

$\boxed{\mathbf{value}\,(e)}$ Value restriction for !-introduction

$$\frac{}{\mathbf{value}\,(p)}\ \textsc{Val\_Prim}$$

$$\frac{}{\mathbf{value}\,(())}\ \textsc{Val\_Unit\_Intro}$$

$$\frac{}{\mathbf{value}\,(\mathbf{true})}\ \textsc{Val\_Bool\_True}$$

$$\frac{}{\mathbf{value}\,(\mathbf{false})}\ \textsc{Val\_Bool\_False}$$

$$\frac{}{\mathbf{value}\,(k)}\ \textsc{Val\_Int\_Intro}$$

$$\frac{}{\mathbf{value}\,(el)}\ \textsc{Val\_Elt\_Intro}$$

$$\frac{}{\mathbf{value}\,(x)}\ \textsc{Val\_Var}$$

$$\frac{}{\mathbf{value}\,(\mathbf{fix}\,(g, x : t, e : t'))}\ \textsc{Val\_Fix}$$

$$\frac{}{\mathbf{value}\,(\mathbf{fun}\,x : t \to e)}\ \textsc{Val\_Lambda}$$

$$\frac{\mathbf{value}\,(e)}{\mathbf{value}\,(\mathbf{fun}\,fc \to e)}\ \textsc{Val\_Gen}$$

$$\frac{\mathbf{value}\,(e)}{\mathbf{value}\,(e[fc])}\ \textsc{Val\_Spc}$$

$$\frac{\mathbf{value}\,(e)}{\mathbf{value}\,(\mathbf{Many}\,e)}\ \textsc{Val\_Bang\_Intro}$$

$$\frac{\mathbf{value}\,(e_1) \quad \mathbf{value}\,(e_2)}{\mathbf{value}\,((e_1, e_2))}\ \textsc{Val\_Pair\_Intro}$$

$\boxed{\Theta; \Delta; \Gamma \vdash e : t}$ Typing rules for expressions

$$\frac{}{\Theta; \Delta; \cdot, x : t \vdash x : t}\ \textsc{Ty\_Var\_Lin}$$

$$\frac{x : t \in \Delta}{\Theta; \Delta; \cdot \vdash x : t}\ \textsc{Ty\_Var}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t \quad \Theta; \Delta; \Gamma', x : t \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let}\, x = e\, \mathbf{in}\, e' : t'} \quad \textsc{Ty\_Let}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash () : \mathbf{unit}} \quad \textsc{Ty\_Unit\_Intro}$$

$$\frac{\Theta; \Delta; \cdot \vdash e : \mathbf{unit} \quad \Theta; \Delta; \Gamma \vdash e' : t}{\Theta; \Delta; \Gamma \vdash \mathbf{let}\, () = e\, \mathbf{in}\, e' : t} \quad \textsc{Ty\_Unit\_Elim}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash \mathbf{true} : !\mathbf{bool}} \quad \textsc{Ty\_Bool\_True}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash \mathbf{false} : !\mathbf{bool}} \quad \textsc{Ty\_Bool\_False}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : \mathbf{bool} \quad \Theta; \Delta; \Gamma' \vdash e_1 : t' \quad \Theta; \Delta; \Gamma' \vdash e_2 : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 : t} \quad \textsc{Ty\_Bool\_Elim}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash k : !\mathbf{int}} \quad \textsc{Ty\_Int\_Intro}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash el : !\mathbf{elt}} \quad \textsc{Ty\_Elt\_Intro}$$

$$\frac{\mathbf{value}\,(e) \quad \Theta; \Delta; \cdot \vdash e : t}{\Theta; \Delta; \cdot \vdash \mathbf{Many}\, e : !t} \quad \textsc{Ty\_Bang\_Intro}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : !t \quad \Theta; \Delta, x : t; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let}\, \mathbf{Many}\, x = e\, \mathbf{in}\, e' : t'} \quad \textsc{Ty\_Bang\_Elim}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t \quad \Theta; \Delta; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash (e, e') : t \otimes t'} \quad \textsc{Ty\_Pair\_Intro}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e_{12} : t_1 \otimes t_2 \quad \Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e : t}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let}\, (a, b) = e_{12}\, \mathbf{in}\, e : t} \quad \textsc{Ty\_Pair\_Elim}$$

$$\frac{\Theta \vdash t' \,\mathsf{Type} \quad \Theta; \Delta; \Gamma, x : t' \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun}\, x : t' \to e : t' \multimap t} \quad \textsc{Ty\_Lambda}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t' \multimap t \quad \Theta; \Delta; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash e\, e' : t} \quad \textsc{Ty\_App}$$

$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \mathbf{fun}\, fc \to e : \forall\, fc.t} \quad \textsc{Ty\_Gen}$$

$$\frac{\begin{array}{c}\Theta \vdash f\ \mathsf{Cap} \\ \Theta; \Delta; \Gamma \vdash e : \forall\, fc.t\end{array}}{\Theta; \Delta; \Gamma \vdash e[f] : t\{f/fc\}}\ \textsc{Ty\_Spc}$$

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \mathbf{fix}\,(g, x : t, e : t') : !(t \multimap t')}\ \textsc{Ty\_Fix}$$

B | Primitives

The following signature gives an indication of how I embedded LT4LA's type system into OCaml's and typed its primitives accordingly. This helped catch bugs and increase confidence in the correctness of the code produced.

```
module Arr = Owl.Dense.Ndarray.D
type z = Z
type 'a s = Succ
type 'a arr = A of Arr.arr [@@unboxed]
type 'a mat = M of Arr.arr [@@unboxed]
type 'a bang = Many of 'a [@@unboxed]
module Prim :
sig
  val extract : 'a bang -> 'a
  (** Boolean *)
  val not_ : bool bang -> bool bang
  (** Arithmetic, many omitted for brevity *)
  val addI : int bang -> int bang -> int bang
  val ltE : float bang -> float bang -> bool bang
  (** Arrays *)
  val set : z arr -> int bang -> float bang -> z arr
  val get : 'a arr -> int bang -> 'a arr * float bang
  val share : 'a arr -> 'a s arr * 'a s arr
  val unshare : 'a s arr -> 'a s arr -> 'a arr
  val free : z arr -> unit
  (** Owl *)
  val array : int bang -> z arr
  val copy : 'a arr -> 'a arr * z arr
  val sin : z arr -> z arr
  val hypot : z arr -> 'a arr -> 'a arr * z arr
  (** Level 1 BLAS *)
  val asum : 'a arr -> 'a arr * float bang
```

```
  val axpy : float bang -> 'a arr -> z arr -> 'a arr * z arr
  val dot : 'a arr -> 'b arr -> ('a arr * 'b arr) * float bang
  val rotmg : float bang * float bang -> float bang * float bang ->
              (float bang * float bang) * (float bang * z arr)
  val scal : float bang -> z arr -> z arr
  val amax : 'a arr -> 'a arr * int bang
  (* Matrix, some omitted for brevity *)
  val matrix : int bang -> int bang -> z mat
  val copy_mat : 'a mat -> 'a mat * z mat
  val copy_mat_to : 'a mat -> z mat -> 'a mat * z mat
  val size_mat : 'a mat -> 'a mat * (int bang * int bang)
  val transpose : 'a mat -> 'a mat * z mat
  (* Level 3 BLAS/LAPACK *)
  val gemm : float bang -> ('a mat * bool bang) -> ('b mat * bool bang) ->
             float bang -> z mat -> ('a mat * 'b mat) * z mat
  val symm : bool bang -> float bang -> 'a mat -> 'b mat ->
             float bang -> z mat -> ('a mat * 'b mat) * z mat
  val posv : z mat -> z mat -> z mat * z mat
  val potrs : 'a mat -> z mat -> 'a mat * z mat
end
```

C | Evaluation Raw Data

Below is formatted output from a trace I obtained by recompiling Owl with print statements inserted on the relevant primitives. I made two modifications: I shortened '_matrix_transpose' to '_mtrsp' for formatting and I split 'posv' into 'potrf/potrs' for a fairer comparison.

```
Chol            Owl             LT4LA           TRANSP
---             ---             ---             ---
empty           empty           empty           empty
_mtrsp          _mtrsp          symm            _mtrsp
empty           empty           gemm            empty
gemm            gemm            gemm            symm
empty           empty           _owl_copy       gemm
gemm            gemm            empty           gemm
empty           empty           _owl_copy       _owl_copy
_owl_copy       _owl_copy       potrf           empty
_owl_add        _owl_add        potrs           empty
empty           empty           potrs           _owl_copy
_owl_copy       _owl_copy       empty           potrf
potrf           getrf           gemm            potrs
empty           getri           empty           potrs
_owl_copy       empty           gemm            empty
_owl_copy       gemm            empty           gemm
_owl_copy       empty           _owl_copy       empty
empty           gemm            symm            gemm
gemm            empty           empty           empty
empty           _owl_copy       symm            _owl_copy
_owl_copy       _owl_sub        _owl_copy       symm
potrs           empty           symm            empty
empty           gemm                            symm
gemm            empty                           _owl_copy
empty           _owl_copy                       symm
```

```
_owl_copy           _owl_add
_owl_sub            empty
empty               gemm
gemm                empty
empty               gemm
_owl_copy           empty
_owl_sub            _owl_copy
empty               _owl_sub
_owl_copy
potrs
empty
gemm
empty
_owl_copy
_owl_add
```

Below is the raw output from the benchmarking script I wrote.

```
##############################################

Alg = CBLAS

Size N Mean (us) Sample    Err+     Err-  R^2
------ --------- ------ ------- -------- ----
     5        24    526       0       -0 1.00
    25       104    370       1       -1 0.98
   125      1803    104      64      -57 0.91
   625    187667   1000   36281   -36281  N/A
  3125  15651064     15  530675  -530675  N/A

##############################################

Alg = LT4LA

Size N Mean (us) Sample    Err+     Err-  R^2
------ --------- ------ ------- -------- ----
     5        41    466       1       -1 0.98
    25       133    343       2       -2 0.97
   125      1678    109      36      -33 0.97
   625    180575   1000   38386   -38386  N/A
  3125  16061291     15  193746  -193746  N/A
```

```
################################################

Alg = Chol

Size N Mean (us) Sample    Err+     Err-  R^2
------ --------- ------ ------- -------- ----
     5        52    448       1       -1 0.98
    25       128    347       1       -1 0.98
   125      1583    112      95      -75 0.74
   625    125526   1000   25502   -25502  N/A
  3125  11210982     15  852463  -852463  N/A

################################################

Alg = Owl

Size N Mean (us) Sample    Err+     Err-  R^2
------ --------- ------ ------- -------- ----
     5        53    444       1       -1 0.97
    25        95    379       0       -0 1.00
   125      1488    116      27      -24 0.97
   625    146150   1000   32346   -32346  N/A
  3125  12108640     15  466381  -466381  N/A

################################################
```

Bibliography

[1] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[2] Philip Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359, 1990.

[3] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In *International Colloquium on Automata, Languages, and Programming*, pages 385–397. Springer, 2013.

[4] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *International Conference on Computer Aided Verification*, pages 781–786. Springer, 2012.

[5] Robert Atkey. The syntax and semantics of quantitative type theory.(2017). *Under submission*, 2017.

[6] John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

[7] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[8] Matthew Rocklin. *Mathematically informed linear algebra codes through term rewriting*. PhD thesis, 2013.

[9] Diego Fabregat-Traver. Knowledge-based automatic generation of linear algebra algorithms and code. *arXiv preprint arXiv:1404.3406*, 2014.

[10] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001.

[11] Henrik Barthels and Paolo Bientinesi. Linnea: Compiling linear algebra expressions to high-performance code. In *Proceedings of the 8th International Workshop on Parallel Symbolic Computation*, Kaiserslautern, Germany, July 2017.

[12] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[13] Henrik Barthels, Marcin Copik, and Paolo Bientinesi. The generalized matrix chain algorithm. *arXiv preprint arXiv:1804.04021*, 2018.

[14] Oleg Kiselyov. The design and implementation of ber metaocaml. In *International Symposium on Functional and Logic Programming*, pages 86–102. Springer, 2014.

[15] Tiark Rompf. Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming. *ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE*, 2012.

[16] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. Partially static data as free extension of algebras.

[17] Troels Henriksen. Design and implementation of the futhark programming language (revised). 2017.

[18] Rust Community. Rust. https://www.rust-lang.org/en-US. Accessed: 08/05/2018.

[19] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Retrofitting linear types, 2017.

[20] Idris Community. Idris 1.2.0 release notes. https://www.idris-lang.org/idris-1-2-0-released/. Accessed: 08/05/2018.

[21] Conor McBride. Code mesh london 2016, keynote: Spacemonads. https://www.youtube.com/watch?v=QojLQY5H0RI. Accessed: 08/05/2018.

[22] Ott. http://www.cl.cam.ac.uk/~pes20/ott/. Accessed: 29/04/2018.